Express Mail Label No. EL068599548US 07-17-00 A

UTILITY PATENT APPLICATION TRANSMITTAL
(Large Entity)

(Only for new nonprovisional applications under 37 CFR 1.53(b))

Docket No.
13679(END9-2000-0080US1)

Total Pages in this Submission
3

jc821 U.S. PTO 07/14/00

jc821 U.S. PTO 09/616809 07/14/00

TO THE ASSISTANT COMMISSIONER FOR PATENTS
Box Patent Application
Washington, D.C. 20231

Transmitted herewith for filing under 35 U.S.C. 111(a) and 37 C.F.R. 1.53(b) is a new utility patent application for an invention entitled:

TEXT FILE INTERFACE SUPPORT IN AN OBJECT ORIENTED APPLICATION

and invented by:

James Richard Wason

If a CONTINUATION APPLICATION, *check appropriate box and supply the requisite information:*

☐ **Continuation** ☐ **Divisional** ☐ **Continuation-in-part (CIP)** of prior application No.: _____

Which is a:

☐ **Continuation** ☐ **Divisional** ☐ **Continuation-in-part (CIP)** of prior application No.: _____

Which is a:

☐ **Continuation** ☐ **Divisional** ☐ **Continuation-in-part (CIP)** of prior application No.: _____

Enclosed are:

Application Elements

1. ☒ Filing fee as calculated and transmitted as described below
2. ☒ Specification having **107** pages and including the following:
   a. ☒ Descriptive Title of the Invention
   b. ☒ Cross References to Related Applications *(if applicable)*
   c. ☐ Statement Regarding Federally-sponsored Research/Development *(if applicable)*
   d. ☐ Reference to Microfiche Appendix *(if applicable)*
   e. ☒ Background of the Invention
   f. ☒ Brief Summary of the Invention
   g. ☒ Brief Description of the Drawings *(if drawings filed)*
   h. ☒ Detailed Description
   i. ☒ Claim(s) as Classified Below
   j. ☒ Abstract of the Disclosure

UTILITY PATENT APPLICATION TRANSMITTAL
(Large Entity)

(Only for new nonprovisional applications under 37 CFR 1.53(b))

Docket No.
13679(END9-2000-0080US1)

Total Pages in this Submission
3

Application Elements (Continued)

3. ☒ Drawing(s) *(when necessary as prescribed by 35 USC 113)*
   a. ☒ Formal Number of Sheets **3**
   b. ☐ Informal Number of Sheets _____
4. ☒ Oath or Declaration
   a. ☒ Newly executed *(original or copy)* ☐ Unexecuted
   b. ☐ Copy from a prior application (37 CFR 1.63(d)) *(for continuation/divisional application only)*
   c. ☒ With Power of Attorney ☐ Without Power of Attorney
   d. ☐ *DELETION OF INVENTOR(S)*
   Signed statement attached deleting inventor(s) named in the prior application, see 37 C.F.R. 1.63(d)(2) and 1.33(b).
5. ☐ Incorporation By Reference *(usable if Box 4b is checked)*
   The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied under Box 4b, is considered as being part of the disclosure of the accompanying application and is hereby incorporated by reference therein.
6. ☐ Computer Program in Microfiche *(Appendix)*
7. ☐ Nucleotide and/or Amino Acid Sequence Submission *(if applicable, all must be included)*
   a. ☐ Paper Copy
   b. ☐ Computer Readable Copy *(identical to computer copy)*
   c. ☐ Statement Verifying Identical Paper and Computer Readable Copy

Accompanying Application Parts

8. ☒ Assignment Papers *(cover sheet & document(s))*
9. ☐ 37 CFR 3.73(B) Statement *(when there is an assignee)*
10. ☐ English Translation Document *(if applicable)*
11. ☐ Information Disclosure Statement/PTO-1449 ☐ Copies of IDS Citations
12. ☐ Preliminary Amendment
13. ☒ Acknowledgment postcard
14. ☒ Certificate of Mailing
    ☐ First Class ☒ Express Mail *(Specify Label No.):* **EL068599548US**

UTILITY PATENT APPLICATION TRANSMITTAL
(Large Entity)

(Only for new nonprovisional applications under 37 CFR 1.53(b))

Docket No.
13679(END9-2000-0080US1)

Total Pages in this Submission
3

Accompanying Application Parts (Continued)

15. ☐ Certified Copy of Priority Document(s) *(if foreign priority is claimed)*

16. ☐ Additional Enclosures *(please identify below):*

Fee Calculation and Transmittal

CLAIMS AS FILED

For	#Filed	#Allowed	#Extra	Rate	Fee
Total Claims	13	- 20 =	0	x $18.00	$0.00
Indep. Claims	3	- 3 =	0	x $78.00	$0.00
Multiple Dependent Claims (check if applicable) ☐					$0.00
				BASIC FEE	$690.00
OTHER FEE *(specify purpose)* _____					$0.00
				TOTAL FILING FEE	$690.00

☐ A check in the amount of to cover the filing fee is enclosed.

☒ The Commissioner is hereby authorized to charge and credit Deposit Account No. **09-0457/IBM** as described below. A duplicate copy of this sheet is enclosed.

- ☒ Charge the amount of **$690.00** as filing fee.
- ☒ Credit any overpayment.
- ☒ Charge any additional filing fees required under 37 C.F.R. 1.16 and 1.17.
- ☐ Charge the issue fee set in 37 C.F.R. 1.18 at the mailing of the Notice of Allowance, pursuant to 37 C.F.R. 1.311(b).

Signature

Dated: **July 14, 2000**

Leopold Presser
Registration No. 19,827

SCULLY, SCOTT, MURPHY & PRESSER
400 Garden City Plaza
Garden City, New York 11530
(516)742-4343

cc:

CERTIFICATE OF MAILING BY "EXPRESS MAIL" (37 CFR 1.10) Applicant(s): **James Richard Wason**			Docket No. **13679(END9-2000-0080US1)**
Serial No. **Unassigned**	Filing Date **Herewith**	Examiner **Unassigned**	Group Art Unit **Unassigned**

Invention: **TEXT FILE INTERFACE SUPPORT IN AN OBJECT ORIENTED APPLICATION**

I hereby certify that the following correspondence:

New Utility Patent Application

(Identify type of correspondence)

is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 in an envelope addressed to: The Assistant Commissioner for Patents, Washington, D.C. 20231

July 14, 2000
(Date)

Mishelle Spina
(Typed or Printed Name of Person Mailing Correspondence)

(Signature of Person Mailing Correspondence)

EL068599548US
("Express Mail" Mailing Label Number)

Note: Each paper must have its own certificate of mailing.

TEXT FILE INTERFACE SUPPORT IN AN OBJECT ORIENTED APPLICATION

Cross Reference to Copending Applications

The disclosure of this application is related to the disclosures of the following copending applications:

"Business Logic Support," serial no. _____, filed _____ (Attorney Docket END9-2000-0079);
"Flexible Help Support In An Object Oriented Application," serial no. _____, filed _____ (Attorney Docket END9-2000-081);
"Dynamic Java Beans For VisualAge For Java," serial no. _____, filed, _____ (Attorney Docket END9-2000-082); and
"Complex Data Navigation, Manipulation And Presentation Support," serial no. _____, filed _____ (Attorney Docket END9-2000-083);
the disclosures of the four above-identified copending applications are hereby incorporated herein by reference in their entireties.

Background Of The Invention

This invention generally relates to text file processing, More specifically, the invention relates to methods and systems for processing text files used to communicate between applications or between an application and an end user.

Much of the communication between applications or from an application to an end user is in the form of text files. Examples are the traditional batch interface files that many "legacy" applications still use, and more contemporary formats such as html, xml and rtf. The latter are examples that indicate that text file interfaces will be with us for a long time.

The problem is that dealing with these files requires a great deal of application logic to parse the incoming text file and do something with the content, or to assemble application data and format it into an appropriate configuration.

There are three basic problems in text file processing. The first is how to describe and program for the structure of the text file. The second is how to achieve a mapping of data between the text file and the application (note that it assumed here that the application has some independent means, in this case EADP, of moving the data from some stored format such as a database). Inherit in this is the third problem: describing the flow of control needed to process the file. The structure of the text file inherently describes the sequence of processing needed to process it; however, organizing this structure into application logic is often a major source of application programming.

Summary Of The Invention

An object of the present invention is to improve text file processing.

Another object of this invention is to provide an improved text file interface support in an object oriented application.

A further object of the present invention is to handle the problems of pattern matching and flow of control, in text file processing, by creating a structure of templates and macros that correspond to the structure of the text file to be processed.

These and other objects are attained with a method and system for processing a text file in a computer application. The method comprises the steps of forming a template from fragments of the text file; using the template as an overlay for parsing incoming files, or as a prototype to generate a segment of an output file; and using a macro class to map data from the text file to an application. The macro class is embedded as a keyword within the template, so that when the template reaches the keyword, the template calls the macro class to further process the text file.

The macro class may be used to read in a segment of the text file and to use the segment to initiate application update processing. Also, the macro class may be used to derive data from the application and to format it into the text file. Preferably, an interface controller is provided to prevent structure clashes by placing text data into appropriate places in a complex object structure as the text file is processed.

This invention is a general solution; so it is not about techniques to produce a specific format (e.g. object serialization or IP format). The only limitations on the text format is that it is possible to describe and predict a mapping from the text form of the data to its application form. Another point that should be noted is that although the application data may be represented by objects, the characteristics of data organization need to process the text files may be completely different.

Since the preferred embodiment of this invention is based on the underlying facilities of the Enterprise Application Development Platform (the copending application "Complex Object Navigation, Manipulation and Presentation" provides a base description), there is an assumption that the application data is of a form that is amenable to EADP control -- that it includes complex object structures and probably some normalized data that can be manipulated using EADP functions.

Further benefits and advantages of the invention will become apparent from a consideration of the following detailed description, given with reference to the accompanying drawings, which specify and show preferred embodiments of the invention.

Brief Description Of The Drawings

Figure 1 is a flow chart outlining a method embodying this invention.

Figure 2 shows major features of a preferred embodiment of the invention.

Figure 3 is a flow chart illustrating an aspect of an embodiment of the invention.

Detailed Description Of The Preferred Embodiments

As mentioned above, there are three basic problems in text file processing. The first is how to describe and program for the structure of the text file. The second is how to achieve a mapping of data between the text file and the application (note that it assumed here that the application has some independent means, for example EADP, of moving the data from some stored format such as a database). Inherent in this is the third problem: describing the flow of control needed to process the file. The structure of the text file inherently describes the sequence of processing needed to process it; however, organizing this structure into application logic is often a major source of application programming.

As generally outlined in Figure 1, the present invention effectively addresses each of these problems. More specifically, this invention solves the first problem, describing the text format, by using fragments of text (templates) as overlays for parsing incoming files, or as prototypes to generate segments of output files. There are several important advantages to this approach. The templates are isolated from all other application logic, so it is easy to see why a particular template is producing a resulting text file. The template is a

literal image of the text fragment it processes, so it is possible to create the templates from samples of the text file. If the format changes, it is easy to change the corresponding template. For incoming files, there is an important advantage to being able to use a template as a mask to parse the text file. This replaces hard to decipher application logic typically used for parsing.

The second problem of mapping data from the text file to the application is solved by specialized macro classes. These come in two flavors (one for input and one for output). The input macro read in a segment of the text file and use it to initiate application update processing. The output macros derive data from the application and format it into the text file (special format classes are used to describe how to transform the output into the proper shape). This invention provides a basic set of macros, and the facilities to add more as needed.

The solution to the third problem, flow of control, is solved by the interaction of templates and macros. A macro is embedded as a special keyword within a template. When the template reaches that point, it calls the macro. The macro in turn is passed another template name as part of its invocation; as a part of its processing it can invoke that template (which in turn calls other macros, etc.). This nested aggregation of templates and macros allows a processing structure to be built up that mirrors the inherit structure of the text file. Since the behavior of the macro depends both on its internal logic and the template it is passed to invoke, it is possible

to reuse the same macro to do different things by passing it a different template. The net effect is that the bulk of the logic needed to describe flow of control is included in the template structure. The template/macro combination is the ultimate expression of the ideal of letting the target file structure determine the structure of the application needed to process it.

The above procedures may produce "structure clashes"; this is a fourth problem that needs to be solved within this context, particularly for incoming text files which must be mapped to application data.

This invention includes an interface controller which sets up a temporary complex object structure in memory. The interface macros pigeonhole data into appropriate places in the structure as the text file is processed. After the entire file is processed, the internal structure is used to process updating data into the application. This master processing module can be used for any EADP compliant application, as can the base interface macros that are provided as part of this invention. The great advantage here is that processing a new batch interface for a new application becomes mostly a matter of setting up the template structure to map out the text file. The macro processing and the base application processing are already in place.

With reference to Figure 2, the preferred embodiment of this invention includes the following major features.

1. Templates that support variable substitution and

conditional or iterative generation for output files;

2. A mechanism to fill in variable substitutions in the templates;

3. A mechanism to handle iterations and conditional processing;

4. A flexible template processing mechanism which operates within an object oriented environment;

5. A flexible and easily extended formatting mechanism for output;

6. A mechanism for template creation and editing;

7. Special support for complex object presentation;

8. Templates that support parsing of the input file and mapping of its contents into a complex object structure suitable for update processing;

9. A mechanism to extract data from the input stream;

10. A mechanism to map the structure of the input file into a complex object structure suitable for internal processing;

11. A mechanism to organize the updating data from the input file and to apply it within the application;

12. A flexible and easily extended formatting mechanism;

and

13. A mechanism for input template creation and editing.

1. Templates that support variable substitution and conditional or iterative generation for output files.

The templates are fragments of text that include the literals for the output text stream. These capture the predictable pattern of the output stream. Each template may include substitution points, which are to be filled in with application data, or calls to macros to perform conditional or iterative processing.

2. A mechanism to fill in variable substitutions in the templates.

A simple variable substitution in the template is indicated by a tag name enclosed by dollar signs. When the template is processed, a Java Hashtable is prepared with the tag names and substitution variables. These are matched against the tag names in the template as the output text stream is created.

3. A mechanism to handle iterations and conditional processing.

The name of a Java class can be used as a substitution variable. If a class name is used, the "macro" method on that class is invoked (it is passed other information which can be included in the substitution variable, such as the name of the next template to process).

Iterations, conditional logic, and the preparation of data for the next template is handled by these "macro" classes.

4. A flexible template processing mechanism which operates within an object oriented environment.

There are two aspects to this.

a. "Macro" classes which all inherit from a common class that defines the "macro" method. There are two base macro classes (one for input and one for output). An interface controller keeps track of the macros needed for a particular process so that they do not need to be continually reinvoked using Class.forName.

b. Java classes which controls template processing. The class for output processing handles functions such as parsing the templates to find substitution variables, resolving the substitutions, and invoking macro classes as indicated in the substitution variables. The class for input processing handles functions such as parsing the templates to find keywords, synchronizing the templates with the input file, preparing a memory structure with the updates, and then applying the updates to the database.

5. A flexible and easily extended formatting mechanism for output.

Much of the format of the output stream is determined by the literal text within the templates themselves. In

addition, the name of a formatting class can be included in the variable substitution name. These formatting classes all inherit from a common class which defines the "format" method. This method can be redefined to do any specialized formatting required for that substitution variable in that template. Of course, the same formatting class can be reused in other places.

6. A mechanism for template creation and editing.

The templates for each application are stored as a Hashtable that is associated to that application's definition class (as described in copending application Complex Object Navigation, Presentation and Manipulation). The template editor is implemented as a Java bean custom editor.

7. Special support for complex object presentation.

A specialized macro is provided which supports the navigation of a complex object structure. This means that only the templates need to be provided to give different types of reports or batch interfaces based on the data in the database underlying the complex object structure.

8. Templates that support parsing of the input file and mapping of its contents into a complex object structure suitable for update processing.

The templates are fragments of text that include the literals for the input text stream. These capture the

predictable pattern of the input stream. Each template includes macros which specify points in the input stream from which data is to be taken (and what to do with that data). The macros can also specify that the field is of a fixed length, so that that number of bytes is automatically advanced in the stream.

9. A mechanism to extract data from the input stream.

An extraction point in the template is indicated by a keyword (a set of characters enclosed by dollar signs). Parameters within the keyword are separated by commas. The first parameter is the name of a macro class. However, the macros used for interface process implement a macro method with slightly different parameters, as described below. The additional parameters give information such as the name of the field to be extracted, which database table it belongs to in the current application, and possibly the length of the input field. Two strategies to recognize the boundaries of the input fields are supported:

a. Some input files use separators between fields and the fields have variable lengths. In this case the separator would be a literal in the template.

b. Others use fixed length fields with no separators. In this case the length of the field is specified in the keyword.

10. A mechanism to map the structure of the input file into a complex object structure suitable for internal processing.

The complex object structure of the target application is defined using the customization mechanisms described in copending application Complex Object Navigation, Presentation and Manipulation, Each keyword in the template used for data extraction specifies the column name and the internal class name for the data to be extracted (specialized keywords can be constructed to assign the same data to more than one table or column). This allows a rapid and simple way to map the contents of the input file onto the internal structure of the application. It also gives considerable flexibility, since the contents of one interface record may be reorganized to map to several internal classes (and thus different tables in the target database) or vice versa.

11. A mechanism to organize the updating data from the input file and to apply it within the application.

This invention takes advantage of the customization facilities described in copending application Complex Object Navigation, Presentation and Manipulation. That invention shows how each database table is associated to an "internal class" that controls updates to that table. The parsing information within the template specifies for each field which internal class it should be assigned to, and the database column name for the field. A memory structure is created to organize this data by internal class and by key data. Within this structure, the

internal classes are organized by complex object structure. When the updates are applied to the database, this is done using the same methods that apply data interactively. The complex object structure is followed, so that rulers are updated before subobjects. All the data for subobjects is linked in memory to the data for the ruler, so that if the ruler update fails, update of its associated subobjects can be avoided.

12. A flexible and easily extended formatting mechanism.

This is similar to the formatting strategy for output files, but in reverse. The format class is used to translate the string data from the input field into a suitable format for internal processing. This can include such things as skipping quotes or extra blank in the input file.

13. A mechanism for input template creation and editing.

The Java implementation adds an interfaceDictionary property to the EADPTemplateDefinition class as a place to define the interfaceTemplates for a particular application. This is then customized for the database definition class for a particular database. The templates for that database are used when an interface for that database is processed.

Each record in the interface file must be matched to a unique template. The type of input record for the template can be specified in two ways:

a. The template name can start with a table name. This table name is then specified when the interface file is processed. Only templates that begin with this name will be used.

b. If the record has an identification symbol at the beginning of the record, this can be used to qualify the name of the template used to parse that record. This is useful when the interface file has a mix of record types. An asterisk in the template name indicates that it can be used for any record in the interface file.

The discussion below describes in greater detail several of the above-mentioned features of the preferred embodiment of the invention.

Output text generation

1. Template definition

The syntax of templates is kept very simple so that templates can be easily created from samples of the target output format. A template is a piece of text with imbedded keywords. The keywords are delimited by dollar signs. For example:

text1.. $keyword1$ text2.... $keyword2$... etc.

A template can span multiple lines of text. However, a special keyword $+$ at the end of a line indicates that a new line character should not be added when the template is processed.

When a template is processed, the text outside the keyword is passed along as is into the output stream. The text inside a keyword is evaluated according to the following rules:

a. The text up to the first comma is checked to see if it matches the name of a VisualAge class. If it does, the "macro" method of that class is invoked. The remainder of the text within the keyword (after the comma, up to the dollar sign) is passed as a parameter to the macro method. It is then passed to the receiving method (additional commas may be used to separate information).

For example, the receiving macro can be passed the name of the next template to process using this technique. This is useful when the output stream requires a nested sequence of templates and macros (for example, a report of purchase orders and line items for each purchase order).

b. If the first part of the text is not the name of a class, it should match the name of a variable in the substitution list which was prepared for use by the template when it was invoked. In this case, any text after the comma is assumed to be formatting information:

i. The first piece is the name of the class which will do the formatting (for example PadRightFormat). If this is omitted, no special formatting is done. The data appears just as it would on the list or entry panels.

ii. If there is more data (delimited by a second comma)

this is passed to the formatting class. For example, the length of the field is passed to PadRightFormat.

2. Resolution of variable substitutions in the templates.

When each template is processed, it is presented a Java Hashtable which has as keys the names of the variables (matching the variables names specified within the template). The values are substituted into the template (they may be modified by the specified formatting routine). The variable substitutions are prepared by macros prior to template invocation.

3. A mechanism to handle iterations and conditional processing.

This invention avoids creating another programming language by adding logic constructs within the template itself. The only branch to logical processing is through invocation of macros, which are coded in Java (and can take advantage of the full power of that environment).

Since templates can invoke macros and macros can in turn invoke templates a recursive chain of macros and templates can be used to handle nested structures within the output stream (for example, reporting orders, and within each order, all the line items for that order).

4. Macro classes

Macro classes all inherit from a common class, com.ibm.eadp.macros.EADPMacroBase, and redefine the

instance method **macro**. The parameters passed are as follows:

a. segment

The macro was invoked from with a template because the keyword (text within dollar signs) contained the name of the macro. There may be other text after the macro name, and this is passed as the segment parameter. The various pieces of information in the keyword are separated by commas. Information that is commonly included here is the name of the template the macro should use for its processing. This allows reuse of the same macro to process different templates.

b. currentClass

This is a reference to an instance of a Java class. Messages can be sent to that class to derive data to populate the variable list.

c. varList

This is the variable list (Hashtable) that was presented to the calling template. Data from it may be used by the macro to build its own variable list.

d. outStream

This is the output stream for the text stream that is to be generated (of type StringWriter).

In addition, the macro class has as a property the current controller, which is the instance of EADPTemplateController that is processing the templates (and which invoked that macro).

5. A Java class which controls template processing.

Template processing is controlled by com.ibm.eadp.macros.EADPTemplateController. It is expected that when a new instance of controller is created, it will be assigned a data base definition class (a child of EADPDatabaseDefinition as defined in Complex Object Navigation, Presentation and Manipulation). EADPDatabaseDefinition has an additional property, templateDefinition, of type com.ibm.eadp.macros.EADPTemplateDefinition. This in turn has two properties, interfaceDictionary and templateDictionary, both of type com.ibm.eadp.macros.EADPTemplateDictionary. This class is a child of Hashtable (redefined to give it a custom editor).

When a new instance of EADPTemplateController is created to be used for processing, it is assigned a database definition class (this gives it access to the database) and a template dictionary (typically the templateDictionary property of the templateDefinition for the database definition).

Some methods:

a. getTemplateDictionary

This method returns the property that was assigned when the instance of controller was created. This is a Java Hashtable that has as its keys the template name, and values the template strings.

b. processTemplate (templateName,currentClass, outStream, varList)

this is the first method invoked in template processing. The first parameter passed is the name of the template. This is used as a key against the template dictionary, and the resulting text (along with the other parameters) is passed to processText.

c. processText(text, currentClass, outStream, varList)

This method breaks the text up into lines and passes each line to processSegment. The EADPStringReader is used for parsing support. EADPStringReader implements the upTo(aString) and upToEnd methods, which are equivalent to the upTo: and upToEnd methods supported by the ReadStream class in Smalltalk.

d. processSegment(segment,currentClass,outStream,varList)

This method parses each line for keywords (delimited by dollar signs). Any text outside a keyword is appended as is to the output stream. Any text within a keyword is passed as the first parameter to processKeyword.

e. processKeyword(keyword, currentClass, outStream, varList)

This method parses out the various pieces of information stored in the keyword (these are separated by commas). The first segment is either the name of a macro class, or the key of a variable substitution to be found on the varList. If the segment is a macro name, the macro(trailer, currentClass, outStream, varList) method on the macro class is invoked (passing the remainder of the keyword as the first parameter). The resolution of the name of the macro class requires more care in the Java environment. This is encapsulated in the macroForName method in the controller class. In order to avoid repeated calls to instantiate the same macro class, a cache of ones that have been used already is kept (all macro methods need to be reentrant to support this). This cache is a Hashtable keyed by the macro name. If the macro is not in the cache, the macroForName method on the database definition class that was assigned to the controller is called. The name of the macro that is passed does not include the package name; the macroForName method use Class.forName, first with the package name com.ibm.eadp.macros, then with its own packaged name (this allows each application to add specialized macros to extend the support provided by this invention).

If the segment is not a macro name, the processVariable method is invoked (passing the segment as the first parameter and the remainder of the keyword as the second parameter

f. processVariable(segment, format, outStream, varList).

This method has as its first parameter the key for the variable substitution. This is used to find the value in the varList, which is a dictionary containing the variable substitutions to be used when processing the template.

The second parameter contains formatting information (separated by commas). The first segment is the name of the formatting class. The remainder is passed as the "pattern" to the format method on the formatting class. If there is no formatting information specified (the format parameter is a null string), the EADPBasicFormat class is used. Format classes are resolved using the same strategy as the macro classes (and formatForName methods).

6. A flexible and easily extended formatting mechanism.

All formatting classes inherit from com.ibm.eadp.macros.EADPBasicFormat class and redefine the format(string,pattern, outStream) method. The first parameter here is the string to be formatted. The second parameter may contain additional formatting instructions, separated by commas (each child would add its own logic to parse the formatting instructions). The formatted result is appended to the passed outStream.

The base method just appends the passed string to the outStream without any additional formatting.

7. A mechanism for template creation and editing.

Template definition is provided as a Java bean customization of the EADPTemplateDictionary. The customization follows There is an EADPTemplateDictionaryEditor which inherits from java.beans.PropertyEditorSupport, and EADPTemplateDictionaryDisplay which is the custom editor.

Templates for an application are edited by customizing the database definition class (child of EADPDatabaseDefintion) for that application. As was noted above, this has a new property, templateDefintion. To customize, in the VisualAge visual editor, a bean of type EADPTemplateDefinition must be added to the visual surface, and attached to the templateDefinition property. The property sheet for the bean then shows two features, interfaceDictionary and templateDictionary, which can be customized (both are of type EADPTemplateDictionary, so they bring up the same type of custom editor).

The custom editor (EADPTemplateDictionaryDisplay) has a list of the current keys for the dictionary, an entry field to define a new key name, and a text area to enter the template text. Buttons to update or delete the template are provided, along with buttons to read and write the text body from a file (this allows the templates to be exported or imported).

The EADPTextHelper class defines the readTextFromFile and writeTextToFile methods. These methods are connected to the open file and save file buttons. Standard file dialog beans (java.awt.FileDialog) are used find directory and file names.

The technique to create the Java initialization string and to initialize the string is similar to the one described in the above-identified copending application "Flexible Help Support In An Object Oriented Application." Only two separators are needed (for the key and text of each template). New line markers are added to the generated string at each line of the template so that the generated code is readable; an extra new line marker is also added within the initialization string for each template line so that the initialized text has the proper line breaks.

8. Special support for complex object presentation.

When a macro is processed, the remaining information in the keyword is passed as the first parameter. The macros described here expected the keyword to contain the name of the internal class (as defined by the above-identified copending application "Complex Data Navigation, Manipulation And Presentation Support For VisualAge Java") and the name of the next template to process. The "current class" would be an instance of the ruler class (for example, the ruler class would be for orders and the name of the class in the keyword would indicate line items for that order). The macro classes have a currentController property which is assigned by macroForName as the macro instance is created. This instance is the one that gets the processTemplate call.

The ReportMacro class redefines the macro method. As mentioned above, the first parameter should contain a string which has the next internal class name and the

next template name, separated by commas. The current class should be an instance of a child of EADPApplicationClass. For the first level, the class name of a top object can be passed, along with an the database definition class for the process. If this is done, the managerForName method on EADPDatabaseDefinition is used to find an instance of the class. Otherwise, an instance of a child of EADPApplicationClass (the ruler class) is passed as the current class, and its subManager method is used to find the subobject class (using the current row of the ruler class as the ruler row).

The class name and template name are parsed out of the first parameter. If the class name matches the name of the current class (this would be true for the first level of the process), the current row of the current class (as the first parameter) and the current class (as the second parameter) are passed to the processRow(row, class, template, outSream, varList) method. Otherwise, a new instance of the type of class indicated by the class name is created (this should be subobject of the current class) and is assigned the current class as a ruler. The current row of the current class is used to open the subobject class (this finds all the subobject rows that match the current row of the ruler, e.g. all lines for a particular order). For each row of the subobject class, the processRow method is called (passing the subobject class as the second parameter). In both cases, the template name parsed from the keyword, and the passed varList and outStream are passed as the additional parameters.

The processRow method copies the passed varList into a new dictionary, and the uses the row dictionary for the passed row to add additional entries. The values in the row dictionary are converted to string format using the getStringValue method on EADPPersistentObject, which does the property editor lookup to format the string value. The processTemplate method on TemplateMetaclass is then called, passing the template name, the class (which is now the subobject class), and the new variable list. Combined with a set of templates that call out various pieces of the complex object structure and provide formatting information, this allows navigation down through the levels of the complex object.

Input text processing

All methods are instance methods. To avoid creating many instances of the same macro class, the classes are cached and reused. The setStringValue method on EADPPersistentObject is used to convert the data from string format to internal format, and this is done as the data is applied to the database. Before that, all data is held in string format. The setStringMethod uses the editor dictionary for the row's data manager to find the right property editor to convert the value to object format.

One difference with the data update function described in the above-identified copending application "Complex Data Navigation, Manipulation And Presentation Support For VisualAge Java," is the way a prototype for a new row is created. In the function described in the above-

identified copending application "Complex Data Navigation, Manipulation And Presentation Support For VisualAge Java", a new row is being added to rows for a result set that has already been created, so that the metadata for the result set can be used to determine the data type for the columns. Here, a new row has to be defined before a query has been issued, so database metadata has to be used. Since this is an expensive call, and a typical interface will not involve many different types of rows, the results are saved and reused.

Several classes and methods are described below:

1. EADPInterfaceView

This defines the visual part used to process a batch interface file. It is customized to provide an interface view for a particular application by customizing its database definition property. This class inherits from Panel, and it designed to be included as a visual bean in the interface view for a particular application.

It has the following features.

a. Table name text area

This is used to specify the table name prefix.

b. Text area for the interface file

The interface file is presented in a text area which

allows editing. This can be useful when the file that is sent to be processed has header and trailer records which are not needed as it is being processed. They can be stripped off manually before the interface is processed.

c. File handling beans and buttons.

The EADPTextHelper class defines the readTextFromFile and writeTextToFile methods. These methods are connected to the Open File and Save File buttons. Standard file dialog beans (java.awt.FileDialog) are used find directory and file names.

d. Continuation

This text area is used to specify a continuation character which may be used by the interface templates.

e. processInterface

This method is invoked when the Process Interface button is pressed. The parameters passed are the interface file, the table name, and the continuation character. To begin processing, a new instance of EAPDInterfaceController is created, and it is assigned the database definition property as its database definition. The interfaceDictionary property of the templateDefinition property of the database definition is assigned to the templateDictionary property of the controller (this is how the controller knows which templates to look for). The datastore property of the database definition is used to connect to the database.

Next, the processInterface method on EADPInterfaceController is called to parse the interface and apply the changes.

f. currentDefinition property

This property is set to the database definition for the application when the visual bean is customized. This is how the Java version knows which database to process. The database definition class also has the connection information that is needed to establish a connection to the database to begin processing.

2. EADPInterfaceController

This class has methods that process the interface file.

a. processInterface

This is passed the interface file (as a String), the table name, and the continuation character.

This method controls the interface processing. It creates a Hashtable (the processDictionary) which acts as a storage area in memory for the updating data. This dictionary is passed as a parameter to all the methods (including macros) that are used to process the interface file.

The first step is to break the interface file into records. The continuation character is used to combine lines from the input file into logical records. For each

record, the matching template is determined by calling the templateForSegment method. The record and its associated template are then passed to processTemplate method.

Once the entire interface has been processed, the updates are applied by calling the doApply method. This iterates through the process dictionary and calls apply.

b. templateForSegment

This method is passed the current line of the interface file and the table name specified as the interface was initiated.

It finds the template that will be used to parse the current record. It uses the templateDictionary property (assigned as the controller was initiated) to locate the dictionary of templates for the current application. It then iterates through the keys of the dictionary, looking for ones that start with the passed table name. If a match is found, it then checks the remainder of the key. If it is an asterisk, the template is used without checking it against the input record (an asterisk indicates that the template is to be used for all records for that table type). Otherwise, the remainder of the template name is checked to see if it matches the beginning of the record (this is used if the records start with flags that indicate the record type). The template is then returned to the caller.

c. processTemplate

The parameters passed in are the template that was found as described above, the current record of the interface, and the processDictionary.

This method first creates a scratch pad entry in the dictionary for holding the processing results for the current row. This is a new dictionary added at the key value 'currentRow' (the processDictionary is a nested set of dictionaries, so most of its entries are other Hashtables). This will be referred to as the currentRow dictionary.

Next, the template is broken up into individual lines (for convenience, the template may consist of multiple lines. This allows each field of the record to be described by a line of the template, which makes it easier to describe and understand the record structure. Typically any existing documentation of the record structure will have this format, and this makes it easier to convert that documentation into a template to process the record.

For each line of the template, the processSegment method is called. This will create entries in the currentRow dictionary.

Once all the lines have been processed, the updateDictionary method is called (passing the processDictionary). This will move entries from the "currentRow" dictionary to their final position in the memory structure.

d. processSegment

The parameters passed in are the current line of the template that was found as described above, the current record of the interface, and the processDictionary.

This method finds a keyword in the passed line of the template by looking for text enclosed by dollar signs. The fragment of text between this and the next keyword is then found (this is the trailer). These are passed to processKeyword.

Next, the method looks for any characters outside dollar signs, and advances the position in the interface record to match those characters (the upTo method in EADPStringReader is optimized to do this). This is how separator characters in the interface record are described in the template. The separator is included as literal text after the extraction keyword.

This process continues until the end of the template line is reached.

e. processKeyword

The passed parameters are the keyword, the fragment of template after that keyword, the interface record, and the processDictionary. The macro name is parsed from the keyword (it is separated from the rest of the keyword by a comma), and is used to find the macro class. The macro method is called on the macro class. Typically, the EADPColumnInterfaceMacro would be called at this

point. However, in some cases the same column data needs to be used in several places (for example, a row in the input may correspond to both a ruler and subobject in the target application if the source application was not well structured).

f. macro(keyword, trailer, segment, dictionary) method in EADPColumnInterfaceMacro

This method will be described now to make it easier to understand the flow of control. Keep in mind that other similar macros could have been invoked at this point (for example, the EADPTwoColumnInterface which places the same column data into two different internal classes). The macro invocation is determined by the placement of the macro name in the template, so the template structure is driving the flow of control at this point.

The parameters passed are the remainder of the keyword (which contains parameter information for the macro), the fragment of the template line between this keyword and the next one, the current interface record, and the processDictionary.

The following parameters are contained in the remainder of the keyword, separated by commas:

1. Internal class name

The name of the internal class to process the update. This may be omitted, if a keyword specifying a standard internal class for the template has been provided (this

is done using the EADPTableType macro).

2. Column name

The database name for the column

3. Format

This is the name of the formatting class that will be used to convert the data before it is used. If omitted, the default class EADPInterfaceFormat is used.

4. Format pattern

If provided, this contains additional parameters for the formatting class.

5. Length

If length is included, it is used to determine how much of the input record to read in to get the column data.

If length is provided, it is used to extract that number of bytes from the interface record. Otherwise, the input record up to the trailer fragment is extracted. In both cases, the current position of the interface record is used as the starting point.

If length was provided, that information is used to advance the current position in the interface record.

The data extracted from the interface record is passed to

the format method of the formatting class.

Once the data has been extracted and formatted, it is placed into the memory structure defined by the passed processDictionary. The dictionary entry at the key "currentRow" is accessed. This as a dictionary keyed by the name of the internal class used to hold the data. The entry for the internal class passed in the keyword is found (if none is present, one is created). This entry is a dictionary that holds data for the row (the keys are the column names). The column name passed as a parameter in the keyword is used to add the data just extracted from the interface record. The values are stored in string format (they are converted later during processing in the apply method).

g. updateDictionary

This method moves rows from the "currentRow" dictionary to their permanent position. To find the permanent position, the key for the row needs to be known (this is why this process is delayed until the entire template has been processed against the interface record. The key columns may be positioned anywhere within the interface record, so the entire record needs to be parsed before it can be classified).

The method iterates over the entries in the currentRow dictionary. The keys are internal class names, and the values are row descriptions (dictionaries of column names and values). These entries are passed to the hasKeys method to determine if the key columns were found. If

so, the addRowToMaster method is called to place the row data in the proper place in the memory structure.

h. hasKeys

This method is passed a key and value from the currentRows dictionary. The key is the name of an internal class. That is used to find the complex object node (EADPComplexObjectNode) for that class in the classDirectory property of the complexObjectStructure property of the currentDirectory property of the database definition (this is encapsulated in the nodeForName method in EADPDatabaseDefinition). The keyCols property of the complex object node is then used to make sure that each has a corresponding entry in the passed value (which is a dictionary of column values keyed by column name).

i. addRowToMaster

This method is passed a key and value from the currentRows dictionary. The key is the name of an internal class. The passed value is a dictionary of column values keyed by column name.

The master dictionary structure is a set of nested dictionaries. It follows the complex object structure, with each instance of a ruler object controlling lower level dictionaries of its subobjects. To achieve this, a two level nesting of keys is used. This first level key is the name of the internal class. The second level is the key information for a particular instance of that class (this is stored as a Vector of the string

values for the key in the same order as the keyCols attribute that was used to set it up). For example, if the complex object structure consists of orders and line items, the master dictionary structure might look as shown in Figure 3.

To accommodate this structure, a node class EADPInterfaceNode is used. It consists of two attributes, rowDictionary the dictionary of column values for the row, and subobjectDictionary, a dictionary of EADPInterfaceNodes keyed by the subobject class names.

The dictForRow method is used to find the proper position within this structure. This positions to the entries for the internal class that are underneath the rulers specified by the key information for the row. Next, a check is made to see if there is an entry for key data matching the current row. If not, a new entry is created. The row data at the entry is then updated with data from the passed row value dictionary.

j. dictForRow

The parameters are the row dictionary, the internal class for the row, and the processDictionary.

This method calls the rulersForClass method to build a Vector of EADPComplexObject nodes for the rulers for the internal class. It then iterates through the Vector. To begin the iteration, the current dictionary is set to the processDictionary, and the current class is set to the first entry in the ordered collection (the top level

ruler).

1. The current class name is used as a key in the current dictionary. The corresponding value is a "row key dictionary" of instances of EADPInterfaceNodes, keyed by the key values for instances of the current class (for example, if the top level is Orders, these would be keyed by the order number).

If there is no entry at the key, a new dictionary is created.

2. Once the row key dictionary has been found, the key for the current class (derived from the values in the passed parameter for row data) is used to find an entry that matches the key values for the row being stored. The entry will be an instance of EADPInterfaceNode. If none is found, a new entry is created.

3. The current dictionary is set to the subobjectDictionary attribute of the EADPInterfaceNode.

4 .The current class is advanced to the next entry in the Vector.

When this iteration completes, the current dictionary will be set to the subobjectDictionary attribute of the EADPInterfaceNode for the immediate ruler of the passed parameters (it will remain at the main dictionary if the passed internal class had no rulers).

k. doApply

This method is called after the interface has been read into the internal memory structure. It is passed the processDictionary. It makes the initial calls to apply. The entries in the processDictionary are passed (the key is passed as the internal class name and the value is the dictionary of interface nodes). At this level, the passed ruler class and ruler node are null.

1. apply

The passed parameters are the name of an internal class, a dictionary of EADPInterfaceNode instances, the ruler class, and the ruler row.

The method invokes itself recursively to work down through the nested structure of dictionaries and interface nodes. At each level it iterates through the dictionary values. Each entry is an EADPInterfaceNode. The rowDictionary attribute holds updating information. An instance of the data manager for the internal class is created using the internal class name. If the passed ruler class is null, the managerForName method on the definition class is used. If the ruler class is not null, the subManager method on its data manager is used, passing the ruler row as the second parameter (this will automatically set up the ruler list). Both these techniques pass along the connection information so that the newly created data manager has database access.

The passed dictionary is a row value dictionary, where the keys are key values (the vector of key values converted to string format). For each entry the key

information in the key for the dictionary entry is converted back to a vector of strings (using the stringToVector method) and the is passed to the rowForKeys method on the data manager (EADPDataManager), which is used to find a matching row on the database (if there is none a new blank row is created). If a new row is created, the fullCompoundType method on EADPDAManager is used to find the column names and types (using database table metadata). Since this is an expensive call, the results are cached by row type in the interface controller.

The editors dictionary is also set up at this time, based on the type of the column. Next, the rowDictionary is used to update column values (using the setStringValue method on EADPPersistentObject, which uses the editorsDictionary property of the datamanager to find the right property editor for each column and convert the string value to object value using the property editor). The updates are applied to the database (this is done using the facilities of EADP internal processing so that any verifications that were specified during customization are invoked at this time). If no updating information was provided, and the row was not found, processing stops for this leg because the associated subobjects do not have a ruler. Also, if the apply failed because of errors, processing stops for this leg.

If processing can continue on this leg, the method iterates over the subobjectsDictionary attribute of the EADPInterface node. For each entry, the apply uses the key for the entry (the internal class name of the

subobject), the value of the entry (a dictionary of EADPInterfaceNodes keyed by key values for the subobject) the application class for the data manager, and the row just processed.

3. EADPInterfaceBase

This class is the base macro class for interface macros and defines the macro(keyword,trailer, segment, dictionary, method).

4. EADPColumnInterface

This class was described during the explanation of processing of EADPInterfaceClass. It is used for the majority of column extraction processing.

5. EADPInterfaceFormat

This is the basic formatting class, and it defines the format: pattern: method. At this level, the method just returns the input string unchanged.

6. EADPTableType

This macro class is used to avoid placing the same internal class name in each keyword within a template. It stores the passed internal class name in a work area in the processDictionary, so it can be used as a default.

As mentioned above, the problem of handling text files, both for input and output, has been around for a long

time, and there have been many attempts at tools. The drawback of most of them is that they are very restricted in the types of flies that they can process, and the customization schemes are very cumbersome. Also, these tools tend to be standalone. One major advantage of the tool disclosed herein (at least for EADP based applications) is that it ties in with the application logic already defined in EADP.

One of the most important characteristics of this solution is that it "inverts" the solution -- the problems of pattern matching and flow of control are handled by creating a structure of templates and macros that correspond to the structure of the text file to be processed. This approach allows for a great deal of flexibility, and it breaks down the grand problem of processing the file into more manageable units.

Since pattern matching is done by duplicating the pattern of the text file within a template, it is easy to set up this part of the tool. Often, a sample of the text file can be used as a basis for the templates.

The macros are designed so that they can be reused for various templates.

Another advantage of this approach is that it can be extended and adapted to new situations. What this invention provides is the basic mechanism needed to invoke the templates and macros. The actual templates and macros needed for a particular file structure can be added as needed within the general context of the tool.

Also, this invention does not require the introduction of a new procedural language within the tool to handle flow of control. All the macros are written in Java. So another advantage is that all procedural logic is written in a standard way. Also, enough information is passed to the macro classes so that they have full access to all the power o the EADP internal classes. This means that the macros can do quite a bit of processing if that is required.

The present invention has been implemented in the Enterprise Application Development Platform (EADP). The user manual for this facility is included herein a Appendix A.

While it is apparent that the invention herein disclosed is well calculated to fulfill the objects stated above, it will be appreciated that numerous modifications and embodiments may be devised by those skilled in the art, and it is intended that the appended claims cover all such modifications and embodiments as fall within the true spirit and scope of the present invention.

CLAIMS

1. A method of processing a text file in a computer application, comprising the steps:

forming a template from fragments of the text file; using the template as an overlay for parsing incoming files, or as a prototype to generate a segment of an output file;

using a macro class to map data from the text file to an application; and

embedding the macro class as a keyword within the template, wherein when the template reaches the keyword, the template calls the macro class to further process the text file.

2. A method according to Claim 1, wherein the macro class reads in a segment of the text file and uses the segment to initiate application update processing.

3. A method according to Claim 1, wherein the macro class derives data from the application and formats it into the text file.

4. A method according to Claim 1, wherein the macro class derives a template name from the invoking template and uses that name to invoke a next template to further process the text file.

5. A method according to Claim 1, further comprising the

step of providing an interface controller to prevent structure clashes by placing text data into appropriate places in a complex object structure as the text file is processed.

6. A system for processing a text file in a computer application, comprising:

means for forming a template from fragments of the text file;

means for using the template as an overlay for parsing incoming files, or as a prototype to generate a segment of an output file;

means for using a macro class to map data from the text file to an application; and

means for embedding the macro class as a keyword within the template, wherein when the template reaches the keyword, the template calls the macro class to further process the text file.

7. A system according to Claim 6, wherein the macro class reads in a segment of the text file and uses the segment to initiate application update processing.

8. A system according to Claim 6, wherein the macro class derives data from the application and formats it into the text file.

9. A system according to Claim 6, further comprising an

interface controller to prevent structure clashes by placing text data into appropriate places in a complex object structure as the text file is processed.

10. A program storage device readable by machine, tangibly embodying a program of instructions executable by the machine to perform method steps for processing a text file in a computer application, said method steps comprising:

forming a template from fragments of the text file; using the template as an overlay for parsing incoming files, or as a prototype to generate a segment of an output file;

using a macro class to map data from the text file to an application; and

embedding the macro class as a keyword within the template, wherein when the template reaches the keyword, the template calls the macro class to further process the text file.

11. A program storage device according to Claim 10, wherein the macro class reads in a segment of the text file and uses the segment to initiate application update processing.

12. A program storage device according to Claim 10, wherein the macro class derives data from the application and formats it into the text file.

13. A program storage device according to Claim 10, wherein said method steps further comprise the step of providing an interface controller to prevent structure clashes by placing text data into appropriate places in a complex object structure as the text file is processed.

TEXT FILE INTERFACE SUPPORT IN AN OBJECT ORIENTED APPLICATION

ABSTRACT

A method and system for processing a text file in a computer application. The method comprises the steps of forming a template from fragments of the text file; using the template as an overlay for parsing incoming files, or as a prototype to generate a segment of an output file; and using a macro class to map data from the text file to an application. The macro class is embedded as a keyword within the template, so that when the template reaches the keyword, the template calls the macro class to further process the text file. The macro class may be used to reads in a segment of the text file and to use the segment to initiate application update processing. Also, the macro class may be used to derive data from the application and to format it into the text file. Preferably, an interface controller is provided to prevent structure clashes by placing text data into appropriate places in a complex object structure as the text file is processed.

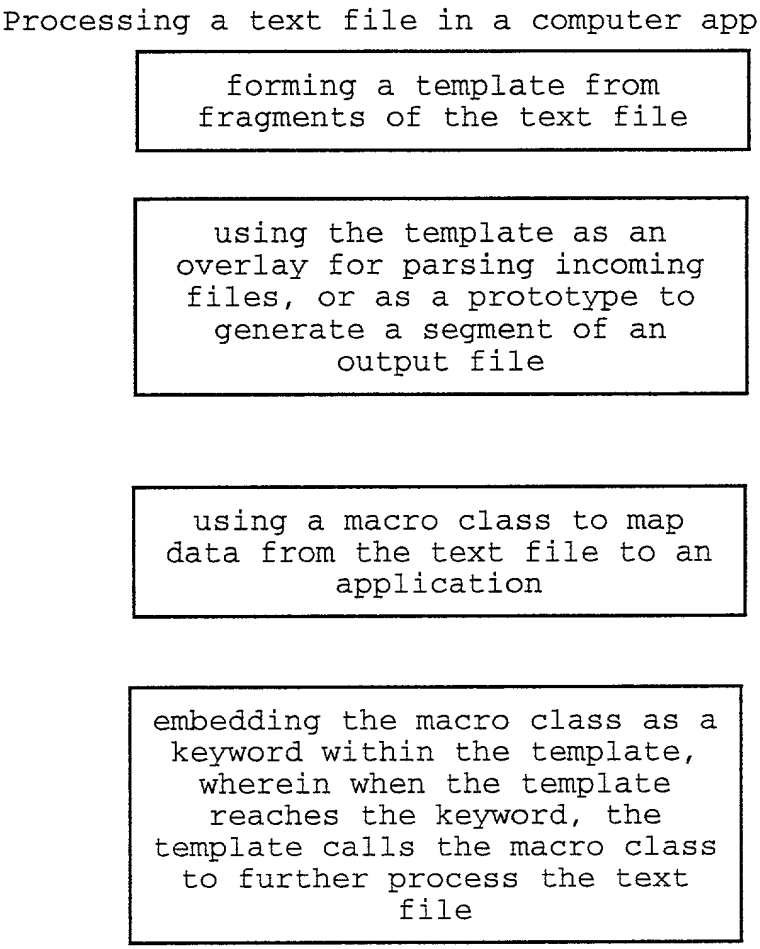
Processing a text file in a computer application
forming a template from fragments of the text file
using the template as an overlay for parsing incoming files, or as a prototype to generate a segment of an output file
using a macro class to map data from the text file to an application
embedding the macro class as a keyword within the template, wherein when the template reaches the keyword, the template calls the macro class to further process the text file

Fig. 1

1. Templates that support variable substitution and conditional or iterative generation for output files

2. A mechanism to fill in variable substitutions in the templates

3. A mechanism to handle iterations and conditional processing

4. A flexible template processing mechanism which operates within an object oriented environment

5. A flexible and easily extended formatting mechanism for output

6. A mechanism for template creation and editing

7. Special support for complex object presentation

8. Templates that support parsing of the input file and mapping of its contents into a complex object structure suitable for update processing

9. A mechanism to extract data from the input stream

10. A mechanism to map the structure of the input file into a complex object structure suitable for internal processing

11. A mechanism to organize the updating data from the input file and to apply it within the application

12. A flexible and easily extended formatting mechanism

13. A mechanism for input template creation and editing.

Fig. 2

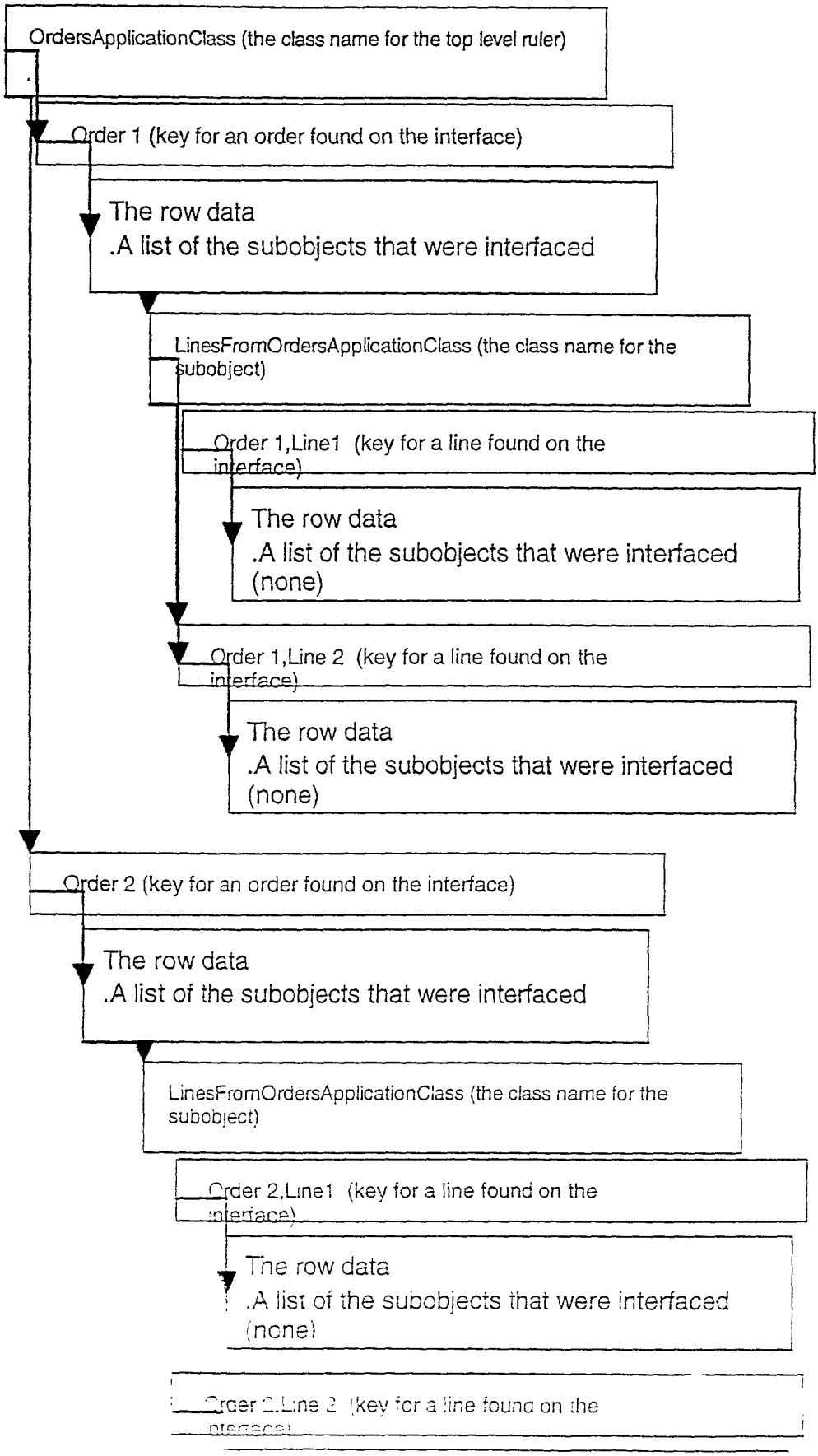
OrdersApplicationClass (the class name for the top level ruler)
Order 1 (key for an order found on the interface)
The row data
.A list of the subobjects that were interfaced
LinesFromOrdersApplicationClass (the class name for the subobject)
Order 1,Line1 (key for a line found on the interface)
The row data
.A list of the subobjects that were interfaced
(none)
Order 1,Line 2 (key for a line found on the interface)
The row data
.A list of the subobjects that were interfaced
(none)
Order 2 (key for an order found on the interface)
The row data
.A list of the subobjects that were interfaced
LinesFromOrdersApplicationClass (the class name for the subobject)
Order 2,Line1 (key for a line found on the interface)
The row data
.A list of the subobjects that were interfaced
(none)

Fig. 3

IBM Docket No.: END9-2000-0080US1
SSMP Docket No.: 13679

Declaration and Power of Attorney for Patent Application

As a below named inventor, I hereby declare that::

My residence, post office address and citizenship are as stated below next to my name; I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled: TEXT FILE INTERFACE SUPPORT IN AN OBJECT ORIENTED APPLICATION

the specification of which (check one)

[X] is attached hereto.

[ ] was filed on _____ as Application Serial No. _____ and was amended on _____.

I hereby state that I have reviewed and understand the contents of the above- identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with Title 37, Code of Federal Regulations, §1.56.

I hereby claim foreign priority benefits under Title 35, United States Code, §119 of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

Prior Foreign Application(s):

Number	Country	Day/Month/Year	Priority Claimed

I hereby claim the benefit under Title 35, United States Code, §120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code, §112, I acknowledge the duty to disclose information material to the patentability of this application as defined in Title 37, Code of Federal Regulations, §1.56 which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

Prior U.S. Applications:

Serial No.	Filing Date	Status

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

As a named inventor, I hereby appoint the following attorneys and/or agents to prosecute this application and transact all business in the Patent and Trademark Office connected therewith: David L. Adour, Reg. No. 29,604, Lawrence R. Fraley, Reg. No. 26,885; Arthur J. Samodovitz, Reg. No. 31,297; John R. Pivnichny, Reg. No. 43,001;William H. Steinberg, Reg. No. 28,540;Christopher A. Hughes, Reg. No. 26,914; Edward A. Pennington, Reg. No. 32,588; John E. Hoel, Reg. No. 26,279; Joseph C. Redmond, Jr., Reg. No. 18,753; Richard L. Catania, Reg. No. 32,608; Leopold Presser, Reg. No. 19,827; Stephen D. Murphy, Reg. No. 22,002; Frank S. DiGilgio, Reg. No. 31,346; Kenneth L. King, Reg. No. 24,223; William C. Roch, Reg. No. 24,972 and Edward W. Grolz (Reg. No. 33,705).

Send all correspondence to:
Richard L. Catania., Esq.
Scully, Scott, Murphy & Presser
400 Garden City Plaza
Garden City, NY 11530

Direct Telephone Calls to: Richard L. Catania, (516) 742-4343

IBM Docket No.: END9-2000-0080US1
SSMP Docket No.: 13679

Declaration and Power of Attorney for Patent Application

James Richard Wason

(1) Inventor: James Richard Wason

Signature: James Richard Wason July 14, 2000
Date

Residence: 32 Spice Bush Lane, Tuxedo, New York 10987

Citizenship: United States of America

Post Office Address: Same as Residence

APPENDIX A

The Enterprise Application Development Platform for VisualAge® Java User Guide

The Enterprise Application Development Platform for VisualAge® Java
User Guide

Persistence Builder Edition
Revised for VisualAge 3.02
James R. Wason

Senior Technical Staff Member

IBM Global Services

Sterling Forest, New York

wason@us.ibm.com

May 31, 2000

Filename = EADPJA3.HTML

Table of Contents

- 1.1.4 Security enhancements
- 1.1.5 Scrolling capability for large databases.

- 1.2 What you need to know about VisualAge®?
- 1.3 Why use EADP?
  - 1.3.1 EADP and Persistence Builder
  - 1.3.2 Build it fast, build it to last

2.0 What is EADP?

- 2.1 Complex Object Support
  - 2.1.1 Primary Complex Objects
  - 2.1.2 Cascaded Complex Object Actions
  - 2.1.3 Quick Views of Normalized Data
  - 2.1.4 Complex Object Support Terms and Concepts
    - 2.1.4.1 The ruler to subobject relationship
    - 2.1.4.2 A database example.
    - 2.1.4.3 Quick Views
    - 2.1.4.4 Complex Object Presentation and Focal Data
- 2.2 Version Control Support.
  - 2.2.1 Formal rules for version control manipulation.
  - 2.2.2 Enhanced Complex Object Support
  - 2.2.3 Isolation of the version control mechanism
  - 2.2.4 Support for relating version controlled objects.
  - 2.2.5 Support for CUA presentation
  - 2.2.6 Version approval and formalization
  - 2.2.7 Audit trails
  - 2.2.8 Version Selection Rules
- 2.3 Business Factor Tables
- 2.4 Derived Fields
  - 2.4.1 Computed fields within a row.
  - 2.4.2 Derived fields for subobjects.
- 2.5 Business Policy Support
  - 2.5.1 Business Rules
  - 2.5.2 Triggers
    - 2.5.2.1 Simple Triggers
    - 2.5.2.2 Compound Triggers
- 2.6 Report and batch interface support
  - 2.6.1 Templates
  - 2.6.2 Macros

3.0 Designing your application to use the EADP

4.0 Importing the EADP Applications.

5.0 Setting up a sample complex object

- 5.1 What you will achieve
- 5.2 Setting up the Persistence Builder Model.
- 5.3 Setting up a test application.
- 5.4 Creating the data base parts
  - 5.4.1 Setting up the database definition class
  - 6.2.2 Customizing the complexObjectStructure property
    - 6.2.2.1 Connecting to the database.
  - 5.4.3 Creating the Customers part
  - 5.4.4 Creating the Item Catalog part
  - 5.4.5 Creating the Orders part
  - 5.4.6 Creating the Line Items part
  - 5.4.7 Finishing touches
    - 5.4.7.1 Column Names
    - 5.4.7.2 Object Names
    - 5.4.7.3 Derived fields
- 5.5 Setting up a default for the Line Item key.
  - 5.5.1 Setting up a new summary column.
  - 5.5.2 Setting up a default editor.
  - 5.5.3 Making the default a special editor for an attribute.
- 5.6 Creating the applet to enter application.
  - 5.6.1 Customizing the application applet.
  - 5.6.2 Allowing the connection to be changed.
  - 5.6.4 Testing the part.
  - 5.6.4 Testing the part.
- 5.7 Creating the list parts for the application.
  - 5.7.1 Creating the list part for Orders.
  - 5.7.2 Creating the list part for Line Items.
  - 5.7.3 Testing the Orders and Line Items list parts.
    - 5.7.3.1 Testing customizations.
  - 5.7.4 Entry Panels.
    - 5.7.4.1 Creating the default entry part for Orders.
    - 5.7.4.2 Creating a custom entry part for Orders.
    - 5.7.4.3 Creating the default entry part for Line Items.
    - 5.7.4.4 Test the entry panels.
  - 5.7.5 Complex Object Actions.
    - 5.7.5.1 Adding a Line Item for an Order.
    - 5.7.5.2 Copying Orders and Line Items.
    - 5.7.5.3 Deleting Orders and Line Items.

6.0 Setting up a version control sample

- 6.1 What you will achieve.

- 6.1.1 Setting up the complex object structure.
- 6.1.2 Setting up the Persistence Builder Definition.

- 6.2 Setting up a version control sample application.
  - 6.2.1 Setting up the database definition class
  - 6.2.2 Customizing the complexObjectStructure property
    - 6.2.2.1 Connecting to the database.
  - 6.2.3 Creating the Affected Item part.
  - 6.2.4 Creating the Item Header part.
  - 6.2.6 Adding Item data to the Affected Item Part.
  - 6.2.6 Adding Item data to the Affected Item Part.
- 6.3 Creating the visual parts.
- 6.4 Creating, Deleting and Undoing Versions
  - 6.4.1 Creating a New Version.
  - 6.4.2 Using the Undo Function.
    - 6.4.2.1 Undo of Modified Data.
    - 6.4.2.2 Undo of Inserted Data.
    - 6.4.2.3 Undo of Extracted Data.
  - 6.4.3 Deleting a Version.

7.0 Batch File Support

- 7.1 Creating Report Templates
  - 7.1.1 Opening the Template Editor for Reports
  - 7.1.2 The Structure of a Template
  - 7.1.3 Creating a Template to Start a Report
  - 7.1.4 Adding a Template for Order Data
  - 7.1.5 Adding a Template for Line Item Data
  - 7.1.6 Testing the Report

8.0 The EADP Help Facility

- 8.1 Enabling EADP help for panels.
- 8.2 Defining the help content.

9.0 Using EADP Interface Support.

- 9.1 Defining Interface Templates.
  - 9.1.1 Mapping templates to records.
  - 9.1.2 Setting up a template to parse a record.
  - 9.1.3 An interface example.
- 9.2 Running interfaces.

10.0 Using EADP Business Rules Support.

- 10.1 Using EADP Business Factor Tables.
- 10.2 Setting up a BFT.

- 10.3 Making the BFT a special editor for an attribute.
- 10.4 Using DynaBean Customization for the BFT.
- 10.5 Setting Triggers
  - 10.5.1 Setting a Simple Trigger
  - 10.5.2 Setting a Simple BFT Trigger
  - 10.5.3 Setting a Simple Summary Trigger
  - 10.5.4 Setting a Compound Trigger
- 10.6 Adding Business Rules
  - 10.6.1 Adding Business Rule Example

11.0 Using more advanced features in EADP.

- 11.1 Restricting Quick View Selection
- 11.2 Specifying strong and weak quick views

12.0 Some more advanced programming techniques.

- 12.1 Extending EADP Classes.
  - 12.1.1 Extending EADPApplicationClass.
  - 12.1.2 Extending EADPDAManager
  - 12.1.3 Extending EADPPersistentObject
- 12.2 Accessing EADP Data.
  - 12.2.1 Accessing EADP Data Manager data.
  - 12.2.2 Accessing EADP Persistent Object data.
- 12.3 Reusing EADP Parts.
  - 12.3.1 Using the EADP Container Part.
  - 12.3.2 Using the EADP Slide Panel.

1.0 Welcome

Welcome to the Enterprise Application Development Platform (EADP). This guide will explain how EADP can help you develop VisualAge® applications more effectively. We will begin with a quick overview that describes some of the services that are available, and how you can use the EADP effectively to design and implement your application. Then we will give some detailed examples which should get you ready to build applications on your own using EADP.

1.1 What's new for VA 3.02?

Those of you who have downloaded previous versions of EADP may be interested in knowing what is different here. If you are new to EADP, skip this section because it won't make sense until you understand more of the terminology.

1.1.1 Enhanced quick view facilities

The 3.02 version introduces many enhancements to take advantage of Persistence Builder relationships; for the most part these are improvements to EADP quick view processing. Quick view relationships can now be navigated just like ruler to subobject relationships (they are presented as usages of the source of the relationship). The notion of "strong" quick view relationships (which are required for update of the target) and "weak" quick view relationships (which are transparent to the user) are introduced. There is also improved capability to describe how to link relationships to provide enhanced navigation -- for example, the "zoom" buttons that allow concatenation of quick view relationships to define quick view columns.

There is also the capability to restrict the selection for a quick view prompt when there is another path to the same target. The other relationship path can be used to restrict which rows are available for selection.

Summary fields can now be defined over a quick view relationship just as they are over a subobject relationship.

1.1.2 String handling in computed fields

Previously computed fields were restricted to numeric data. You can now define string handling computations, and these can be mixed with numeric computations.

1.1.3 Calculated keys

It is now possible to add a custom editor that will default the key value in a new row to the "next" key within a defined sequence. This works when the row is added within the constraints of a bound relationship (a subobject or string quick view relationship). An example would be the "next" line number for a line item in an order. Note that the key can be string or numeric.

1.1.4 Security enhancements

EADP internal processing now allows a "session id" to be passed through. This can be used to restrict which objects are presented to the user.

1.1.5 Scrolling capability for large databases.

EADP has always allowed the restriction of query results by means of selection criteria. What is new here is the capability to limit the results of unbounded queries. The maxrows feature in JDBC is used to limit the number of row returned; in addition, an order by clause is added to the selection to make sure results are returned in key order. A restart capability is also provided to select the "next" set of rows; this permits controlled scrolling through a large database.

1.2 What you need to know about VisualAge®?

The version of EADP described here is an extension of the Persistence Builder features of VisualAge® for Java. It provides enhanced support to create sophisticated business applications using visual programming. EADP also adds some flexibility to Persistence Builder. It allows a selection string to be specified to limit database selection, and it allows the database connection to be specified at runtime. EADP protects against unbounded queries by limiting the number of rows processed by the query, and it provides restart logic to allow scrolling through a large database. In addition, EADP provides visual parts that allow tabular update.

Before reading about EADP or attempting the examples, you should be familiar with some of the examples in the Persistence Builder user guide. You will find that EADP makes the programming involved considerably simpler. However, you should understand how to create models and schema within Persistence Builder.

Sample databases are included in this package which allow you to use either DB/2 or an ODBC database as the sample database. If all else fails you should be able to use the dBase 5 driver included as a part of the ODBC package. Most features will work using the image persistence feature of Persistence Builder; however, if you do this you will not be able to use the prebuilt sample databases.

1.3 Why use EADP?

EADP is the next generation in visual programming. It lets you design complicated business applications using visual design techniques. Using EADP, you can do rapid prototyping and iterative development on applications that were too large for other tools to handle. This greatly increases the chances that your application development will succeed.

EADP is integrated with the database design environment provided by Persistence Builder, and adds additional models to control presentation and business logic. The combination provides a total development environment that allows the entire application to be model driven for both design and execution.

EADP adds some vital features to Persistence Builder to allow the proper handling of large databases. It provides convenient wrappers for Persistence Builder transaction management and relationship management so that you don't have to write your own code to take advantage of these features.

EADP is based on many years of experience in developing large scale data base applications. It pulls out function that has traditionally been considered application specific, and lets you add it to your new application using the same visual programming techniques as the rest of the VisualAge® product. Some examples are:

1. Centralization of design and implementation. EADP provides a consistent context for all the

features you need to produce a complete working application. In EADP, a design element is defined one time in one place, and is consistently applied throughout the application (examples of this are database verifications, external names, version control, and complex object rules). Each of these design elements is isolated and can be changed independently. The EADP toolset gives an organized presentation of these design elements so that you can quickly create them, and continue to control them once they have been created.

2. Linking together data in different tables. For example, if customer number is part of the purchase order table, providing customer name and address with the list of purchase orders.

3. Providing transitions from one table to another, for example from a list of purchase orders to the list of lines for a selected purchase order. Any navigable relationship defined in Persistence Builder can be used to define a data navigation in EADP. In addition, EADP provides some capability to join relationships to provide enhanced navigation.

4. Adding derived fields to a list. For example, if the purchase order line includes unit price and quantity. including the total price of the line item (unit cost times quantity) as a derived field. EADP can provide both numeric and text based computed fields, and it can mix and match numeric and text conversions within one computation.

5. Providing defaults for fields. The same mechanism used for computing derived fields can be used to provide default values that are based on related data. This can be particularly useful for providing the value of the "next" key based on the maximum of existing keys that are bound together by a relationship.

6. Adding derived fields based on calculations over a related table. For example, the total cost of a purchase order, which is calculated as the sum of the total price of the lines for that purchase order.

7. Adding business meanings for the data. EADP provides two support mechanisms, Business Factor Tables and triggers to do this. These allow you to define date states that are used to drive decisions within the application.

8. Organizing and isolating business policies and rules. Verifications can be isolated from the rest of the application. Many verifications can be defined completely using visual programming techniques.

9. Isolating informal data. EADP version control gives you the capability to define persistent

units of work, and to keep an audit trail of changes to your data over time.

10. Generating reports and batch interface files. EADP template processing allows you to quickly define report and batch file layouts.

11. External name support. EADP allows central definition of the external names of data elements. This includes usage in titles for list columns and entry fields, the names of your tables when displayed to the user, and the displayed value for data elements with discrete values.

12. Server support. Although EADP comes with a visual interface based on applets, it is fully configured for server side support. EADP provides well defined and tested interfaces for server side application development. One advantage of this is that the applet interface can be used during development to test the EADP based applications. EADP then provides adapters (not included in this package) to hook other interfaces into the server side code.

1.3.1 EADP and Persistence Builder

The Persistence Builder edition of EADP uses Persistence Builder facilities for database modeling. This includes defining the relationships that are important for EADP (these are the "quick view" and "ruler to subobject" relationships). For the most part, EADP function interacts with Persistence Builder at the model level, so that EADP doesn't care how that model is implemented. For example, you can use Persistence Builder image persistence instead of the the supplied sample databases to do the samples (of course, you will have to load in the data by hand). However, there are some advanced features, such as the use of the target field name for quick view relationships, which depend on the sql code generated by Persistence Builder. These have been tested for DB2, but may not necessarily work for other database systems.

The application layer of EADP includes logic to handle Persistence Builder transaction levels. The "quick view" facilities provide an easy way to implement normalized data using Persistence Builder associations. EADP complex object support works within the context of Persistence Builder one to many relationships. Finally, EADP version control provides a way of mapping "fuzzy" relationships within Persistence Builder, and it provides complete application support to make this easy to use within applications.

EADP visual support provides list parts that support tabular update (these are resizable and also support split panel capability). EADP also provides support for focal data, and both canned and very customizable entry panels.

1.3.2 Build it fast, build it to last

EADP lets you build a working prototype with complex panel transitions quickly and effectively.

More important, it gives you a framework to build an application that is easy to understand and maintain. It does this by allowing you to centralize and control data definitions, external names and the implementation of business policies. All of these are effectively isolated so that they can be changed without breaking the rest of the application. One final benefit is that EADP provides a well defined context in which to design your application, so the process of translating requirements into implementation becomes greatly simplified.

2.0 What is EADP?

EADP provides run-time services which your application can invoke instead of reinventing them. It also provides build time services to allow you to build your application visually using the Visual Composition Editor. These services work together in an integrated manner; however, for convenience, we have broken them out into several components.

2.1 Complex Object Support

Some very simple data base applications use only one type of data base table to store their data. Usually, however, several different types of tables are required. The application then needs to present the data from these multiple tables in a way that allows the user to navigate easily through the data.

2.1.1 Primary Complex Objects

The ORDERENT database that was shipped as a sample with VisualAge® Smalltalk Version 2 is a fairly typical example (this is included as one of the sample databases here). It has four tables: CUSTOMERS, ORDERS, LINE_ITEMS, and ITEM_CATALOG. The LINE_ITEMS table has line items for the purchase order (this is an example of a one-to-many type data relationship that forces use of a separate table). Although it is possible to present all the line items for all the orders in one big list, it makes more sense to select an order first (from a list of selected rows in the ORDERS table), and then use this to select a list of lines for that order. The purchase order, which is the collection of the data in ORDERS and LINE_ITEMS table (for a particular order number) is a small example of a "primary" complex object relationship.

2.1.2 Cascaded Complex Object Actions

Once you have set up a primary complex object, the delete and copy actions can cascade through the complex object structure. The same mechanisms that are used to select subobjects for presentation are used to select the subobjects to process for the cascaded actions.

2.1.3 Quick Views of Normalized Data

More subtle relationships are to be found among the other tables. For example, the ORDERS table includes CUSTOMER number. However, a user view of the purchase order might want to show additional columns from the CUSTOMERS table, such as name and address information. The "quick view" facility of EADP allows you to include this normalized data using visual programming techniques.

2.1.4 Complex Object Support Terms and Concepts

Most applications need to deal with panel transitions and normalized data, but until now the logic to do it has been regarded as application specific. EADP abstracts several behaviors which are common to all complex objects. In order to talk about this, we need to introduce a few new terms and concepts:

2.1.4.1 The ruler to subobject relationship

A ruler to subobject relationship is a special kind of one to many relationship where the "one" object controls the "many" object (for example, a purchase order header controls the line items for the purchase order). This is the relationship that glues together complex objects. When an object processes, it does so in the context of its rulers. Within Persistence Builder, each type of class needs to have a primary key. For a ruler to subject relationship to be in place, the primary key of the ruler must form part of the primary key of the subobject. When this is modeled in Persistence Builder, the "ruler" relationship becomes one of components of the subobject key. In more complicated applications this can extend to an entire hierarchy (the same table may be both a ruler and a subobject).

2.1.4.2 A database example.

The ORDER_NUMBER column (in both ORDERS and LINE_ITEMS) is a good example of a key column. The value in the key column is used to glue together the complex object relationship between the two tables.

From a database perspective the logical key is a set of key columns which uniquely identifies a particular occurrence of an object. Within a "primary" complex object, the columns of a ruler provide a partial primary key of its associated subobject. For example, the physical key of the Purchase Order Header is ORDER_NUMBER, and the physical Key of the Purchase Order Line is ORDER_NUMBER/ORDER_LINE_NUMBER.

When this is modeled and mapped within Persistence Builder it may be a little less obvious. Suppose that onumb is mapped to the ORDER_NUMBER column of the ORDERS table. Originally onumb is mapped to ORDER_NUMBER and lnumb to ORDER_LINE_NUMBER in the LINE_ITEM table. However, when a ruler -> subobject association is set up, the "ruler" relationship in the Lines model takes the place of onumb. So the primary key is now ruler/lnumb (the example for the ORDERENT database will do this in detail).

EADP function uses a naming convention to pick out which Persistence Builder relationships are ruler to subobject. The name of the "ruler" role should always be ruler. A subobject can only have one ruler, so this safe.(In addition, as noted above, ruler should form part of the primary key of the subobject). A ruler can have many subobjects; for each, the role name should begin with subobject. The standard names that you can use in the Java version are subobject1, subobject2, subobject3, etc.

2.1.4.3 Quick Views

A quick view is another type of one to many relationship that handles normalized data needed for lookup. For example, references to a customer object from within an orders object. The "one" side of the relationship (the customer) is not the ruler of the order. However, the data for the customer needs to be accessible from within the order.

Physically, quick views are implemented by mapping the relationship to the primary key column in the source table (the Customer), and to a foreign key (the quick view column) in the target table (the Order). This mapping is done within Persistence Builder. EADP then allows you to pull in data from the customer (such as name, street, etc.) and include it with the visual presentation and application logic of the Order. To distinguish quick view relationships, the source role should begin with qvsource (qvsource, qvsource1, etc.). The reverse role should begin with qvtarget, and use the same suffix as its associated source.

You can also use quick view relationships for data navigation, if you set up the relationship correctly in Persistence Builder. The default list panels provided with EADP include the ability to navigate through both directions of a quick view relationship. The "source" side of the relationship is viewed using the "quick open" facility. The "target" side is presented as a usage of the source.

2.1.4.3.1 Strong and Weak Quick Views: The 3.02 edition of EADP introduces the motion of "strong" and "weak" quick views. A "strong" quick view is almost like a ruler to subobject relationship. The difference is that for a ruler to subobject relationship the ruler relationship is part of the key for the subobject; this is not necessarily true for a "strong" quick view. If an object is the target for a strong quick view, it cannot be created except from a list of usages for the source relationship -- this is similar to the rule that a subobject must be created within the context of its ruler. This allows key values to be determined from the maximum of the keys within the context of the strong quick view relationship. An example would be to calculate the next order number for a particular customer based on the maximum of the order numbers so far for that customer.

Weak quick view are used to provide header records for summary calculations. Weak quick views can be made transparent within the application so that they do not impede user creation of data. The "source row" for the weak quick view is system created.

2.1.4.4 Complex Object Presentation and Focal Data

You can visually create the subobject list panel. It is automatically connected to the list panel for its ruler. On the subobject panel you can visually add "focal data" (this is data about the ruler). This is automatically filled in from the selected row for the ruler as the subobject panel is opened. EADP provides a focal data form, and list and entry panels for subobjects that incorporate it.

2.2 Version Control Support.

Most applications need version control. Quite often the initial descriptions of an application may not specifically point this out. However, there are several common application requirements which can be satisfied within the context of the version control support provided here. These are:

1. The need to segregate work in progress from completed and approved data.

This requirement is often expressed as the need for a "private cache" for a particular user. The version control support provided by EADP can isolate informal versions (work in progress), from formalized versions.

2. The need to keep a historical record of data.

The version control support in EADP keeps back levels of data. It does so selectively (only the data that is actually modified is copied to retain the previous information).

If you already have specific requirements for version control, you should be able to fit them into the version control mechanisms that EADP provides. EADP version control support isolates properties common to any version control mechanism, and provides a convenient way to plug in the application specific logic. One advantage of this approach is that as your version control requirements evolve, it is likely that you can isolate the changes without affecting the rest of the application. If you originally developed your application without version control, you can add it in later with a minimal amount of effort (as long as the application was developed using EADP).

EADP provides a default version control mechanism which will be useful for many applications. Named versions are controlled by version selection objects (VSOs). Each version selection object contains the name of the version, and the logical key of data that is version controlled (typically the top level of a complex object). A named version acts as a persistent unit of work; formalizing it is like a unit of work commit. Up to the point of formalization, all the updates done by that version can be rolled back (using the undo function).

EADP Version Control Support consists of:

2.2.1 Formal rules for version control manipulation.

In order to provide generic support for version control, certain formal behaviors must be isolated

and abstracted. These are the concepts of logical key (which identifies associated data "up to" version selection), and selection sequence number. The data base columns which are part of the logical key are identified as within Persistence Builder as a "root" class for table. The primary key for the table class adds sequence number control which allows for multiple versions. There are two sequence numbers include as attributes; an insert sequence number which is part of the primary key, and an extract sequence number which is used along with the insert sequence number to control version selection. A specialized relationship (root to version) is used within Persistence Builder to associate the root to its versions (when this is modeled, the primary key of the versioned table becomes the root relationship plus the insert sequence number). EADP functions uses Persistence Builder generated queries to find the root and all its versions, and then uses its own application function to select a version associated to that root class (EADP processes this entirely at the model level, so that version control can be implemented using Persistence Builder image persistence).

2.2.2 Enhanced Complex Object Support

EADP Version Control Support is compatible with Complex Object Support. The ruler -> subobject relationships are mapped root to root if the classes have a root -> version relationship. In addition, there are several new complex object actions (promote, promote verify, and undo), which are cascaded.

2.2.3 Isolation of the version control mechanism

The version controlled object understands versions only in terms of the insert and extract sequence. The interpretation of these must be application specific. However, EADP gives you a set of methods to get you started. It also provides support for a common type of version control mechanism (version selection objects) so that you can get this up and running just by setting up the data base tables and Persistence Builder model.

2.2.4 Support for relating version controlled objects.

One of the problems with version control is that relationships get "fuzzy". An initial entity model may show that "A" is related to "B". However, once we introduce version control, "A" may have versions, "B" may have versions, and the relationship itself (if there is intersection data) may have versions also. Now, which version of "A" is related to which version of "B", and with what intersection data? There are some standard rules for sorting this out, and they are provided as a part of EADP.

These rules are used for selecting objects for quick view and derived (subobject) fields, so that the version control mechanism is fully integrated with the extended features of complex object support.

2.2.5 Support for CUA presentation

The CUA presentation of version controlled objects is an extension of the CUA support provided for complex objects. This provides support for updates of version controlled objects (to ensure that an appropriate version is available for update when a version controlled object is opened from an entry or list panel). Presenting the top layer of a version controlled application is particularly tricky, since you need to be able to show all the versions of the top level level objects in the complex object hierarchy. EADP provides mechanisms to provide transitions (using visual programming in the Composition Editor) from a top level list (all versions of several master objects) to a subobject list of data related to one master object at one version.

2.2.6 Version approval and formalization

A version of data can be informal (it can be updated but not trusted because it may change) or formal (it can be trusted because it has been approved, but it can no longer be updated). EADP Version Control Support provides mechanisms for segregating formal versions from informal versions, for blocking updates when the version has been formalized, and for progressing a version from informal to formal status, or removing an informal version if it is not approved.

2.2.7 Audit trails

Insert/extract control is used to handle multiple versions of data for a complex object. This allows the complex object to be presented or updated at the various versions, without requiring that all the data for the complex object be copied to each version. All the back version can be accessed to provide an audit trail of how the data change over time.

2.2.8 Version Selection Rules

EADP uses Persistence Builder function to find all versions related to a root, and then selects the proper version or versions based on the following rules (in most cases a selection sequence is passed).

1. Applicable

This selects the version with insert sequence less than or equal to the selection sequence and extract greater.

2. Affected At

This is like the Applicable, but it also includes data that was extracted by the version.

3. All Versions

This finds all rows which match the logical key.

4. VSO Selection

The version selection object is associated to the root of the top object by a "reverse" ruler -> subobject relationship (the VSO plays the role of ruler, but its primary key includes subobject as a component). In this one case, the ruler is on the "many" side of the one to many relationship.

2.3 Business Factor Tables

A business factor table is a mechanism to isolate application decisions from the specific values of data elements. The value of the data element is used as the key to a row of the table. Application decisions are based on values in the corresponding columns. Business factor table support is fully integrated with other EADP functions. A new data type (coded data element) has been introduced for data elements with discrete value. Customization of coded data elements allows you to attach them to business factor tables, and to edit the business factor tables to adjust the row and column values. The business meaning of the particular column (or column value) can be captured in a trigger (see 2.5.2, "Triggers"). BFT's provide enhanced external name support to display the value of the data element using an external (meaningful) name.

2.4 Derived Fields

Much of the information that is of interest in an application is not stored directly on the database; instead it has to be derived or computed from the raw data. Business factor tables give one example of how EADP allows you to interpret data into user terms, and to store the business meaning of that data. Derived field support gives you the additional ability to present and manipulate derived data as if it were stored with the rest of the row on the data base. These derived fields can then be used to drive application logic.

2.4.1 Computed fields within a row.

EADP allows you to specify formulas to compute fields from data within one row. The fields that can be used in the calculation include other computed fields. For example, the line item includes a column for unit price and a column for quantity. A computed field can be added to multiply these, giving the total cost of the line item.

With EADP for VA 3.02, the ability to mix and match text and numeric calculation for computed fields is added. This includes the ability to split text at a certain substring, convert the remainder to a number, do numeric calculations with it, and convert the results back to text.

2.4.2 Derived fields for subobjects.

EADP also allows derived fields to be calculated for all rows of a subobject for a particular row (for example, all line items for an order), or for all usages of a source column in a quick view

relationship). This can be used to give the total cost of the order as a computed field. The formulas that are supported are all, sum, first, last, average, minimum, maximum and count. Any of these can be restricted to rows matching selection criteria specified by a simulated sql statement. Since this is processed as an internal trigger, the columns used for selection do not have to be restricted to true database columns; they can include other derived columns as well.

2.5 Business Policy Support

Business policy support allows you to control the implementation of business rules within your application. It encourages the isolation of these rules from each other and from the rest of the application, and it provides an efficient and convenient mechanism to implement these rules.

2.5.1 Business Rules

A business rule is an executable part of the application. Each rule is defined by the following:

1. The Persistence Builder class (database table) it acts upon.
2. The VisualAge® class that implements the rule.
3. The event that causes the rule to be invoked. These are:
    1. Data creation
    2.
    3. Data modification
    4.
    5. Any data update (create or modify)
    6.
    7. Data deletion
    8.
    9. Data formalization
4. The error message to be issued if the rule fails
5. Any triggers to be used to check if the rule should be invoked

Rule definition allows an implementing method to be defined. This is optional. If the rule is completely defined by its message and triggers, you don't need to do anything else. However, if you need to do more you can create a method to add any extra logic that is needed to fully implement the rule. This approach encourages the isolation of the rules logic from the rest of your application, and in many cases it will allow you to implement rules entirely through visual

programming.

2.5.2 Triggers

A trigger is an evaluation of whether or not the application data meets a certain condition or set of conditions. EADP allows you to define and document triggers, and to use them to drive business logic. It also provides a caching mechanism to allow for the efficient processing of triggers during runtime.

2.5.2.1 Simple Triggers

A simple trigger checks the value of a single attribute in a single table Persistence Builder class (if the attribute is supported by a BFT, it will check the value of a BFT column for the row that matches the value in the database column). EADP provides an external interface to define simple triggers. This includes documentation of what the trigger means. Simple triggers can use quick views and derived fields to make decisions based on complex business information.

2.5.2.2 Compound Triggers

A compound trigger is made up of simple or compound triggers. As each trigger is added, it can be evaluated as either true or false to provide the compound result. The set of triggers is then evaluated using "and" (all satisfy the true/false condition for the individual trigger) or "or" (at least one of the triggers satisfies its condition). When a compound trigger is evaluated, the evaluation stops as soon as the compound result has been determined.

2.6 Report and batch interface support

EADP uses a system of macros and templates to generate reports and batch interface files. A special macro is provided which allows you to conveniently write reports that include an entire complex object structure.

This same facility can be used in reverse to parse incoming batch files and use them to update databases controlled by EADP applications. In this case the template is a mask that shows how to parse the incoming records and map the data to fields in internal classes defined to EADP.

2.6.1 Templates

A template is a fragment of text with substitutions. The substitution areas are delimited by dollar signs. For report generation, you can:

1. Put an attribute name as a substitution name. This will place the value of that attribute (for the current row) in the text at that point. The external value will be used. You can add formatting information such as alignment after the name of the attribute. The attribute can be an actual

data base column, a quick view column, or a derived or computed field.

2. Indicate that you want to include data for a subobject at this point in the report. The substitution includes the name of the macro to process the subobject (which usually is the standard one provided by EADP), the name of the subobject's internal class, and the name of the template to be used to process the subobject.

For incoming batch processing, the macro name is the name of a macro that derives data from the input record and maps it to a particular field controlled by an EADP internal class (see 9.0, "Using EADP Interface Support.").

2.6.2 Macros

Macros are special VisualAge® classes which are used within template processing to gather data or to decide what to do next. A standard report macro is provided that finds all the subobjects (for the passed subobject type) that match the key of the current row. It then processes the passed template against the subobject row. This macro can be successively invoked by a series of templates to report an entire complex object structure.

3.0 Designing your application to use the EADP

The function provided in the EADP is an extension of the data base support provided by VisualAge® Persistence Builder. One important aspect of this is where "persistent" data is defined. Persistence Builder provides code generation facilities to create the model and service class needed for persistence support. EADP takes full advantage of this. The implication is that you should set up your Persistence Builder model and generate out the persistent services first. The model definitions for complex object support and version control need to be defined with Persistence Builder (using the correct naming conventions).

EADP also influences the way you set up your data base design. It is important to work out the transitions from table to table (this is what turns into the definition of the complex object structures). Once you have established this, make sure that the "key" columns are consistent where ruler -> subobject and root -> version relationships need to be mapped.

Once you have done this, you should be able to rapidly put up a shell of your application (using visual programming support provided by EADP) that will allow you to select and update the various tables in your application, and navigate through that data.

EADP also provides a well defined mechanism for implementing business rules. To take full advantage of this, you should be able to relate data conditions to business decisions. This will

allow you to set up the necessary business factor tables and triggers. The triggers should become a vocabulary to describe your application in business terms. As you build rules, you should reuse the same vocabulary as much as possible.

EADP encourages you to make each business rule small and isolated. If you are using a top down approach, you can define the policies first, then the rules. However, it is probably better to define the triggers "bottom up" because typically the same trigger will be used in many different rules. If you don't understand the application well enough to do this in the beginning, keep in mind as you implement new rules that you should try to use existing triggers.

4.0 Importing the EADP Applications.

The file that is provided in the installation package is a self extracting zip file eadpjava.exe which includes the interchange files eadpjava.dat. This includes three projects that you need to import into your IDL (EADP Access Classes, EADP Views, and EADPSamplesProject). You need to load these into the Workspace. Make sure that you have loaded the IBM Persistence Builder EJB Library, the Visual Persistence and the VisualAge Persistence Common Runtime projects before attempting to load the EADP projects.

If you are operating at VAP levels other than 3.02, you will see that there are some errors in EADPDataStore (the signature of some of the abstract methods has changed from release to release of VAP). The corrections you need to make should be self evident; none of these methods are used by EADP, so they just need to be defined properly for your particular version of VAP.

The EADP projects are structured so that EADP Access Classes has no references to AWT classes and can be used for server side code. (Unfortunately, the VAP classes have a Swing dependency so they cannot be isolated quite so cleanly; however the use of these classes with server side EADP will not start an AWT thread on the server).

5.0 Setting up a sample complex object

The first example uses the sample data base ORDERENT which was shipped with the VisualAge® Smalltalk product in Version 2. You can install either the DB2 or dBase 5 (ODBC) version of this database.

The installation program assumes that you have already created the database.

1. To create the ODBC version of this database:

    1. You should have the dBase 5 ODBC driver available as a part of the VisualAge® package. (Note that this was not shipped in recent versions of VisualAge).

2.
3. Create a new directory x:\orderent (where x is the drive of your choice).
4.
5. In the ODBC administrator, configure ORDERENT as a dBase 5 data source with the orderent directory as the data source.

2. To create the database in DB2, issue the following line from a a DB2 command prompt (if you are not using UDB)

```
create database ORDERENT
```

If you are using UDB, you must create the database using the the Control Center.

Before you try to load the sample data, make sure that you have followed the instructions in the VA user manual to enable the database driver you will be using (the jdbcodbc driver or the db2 app driver).

Now run com.ibm.eadp.samples.EADPSampleDisplay. Choose the type of database you are loading, and press the Create button.

Make sure you have this data base installed, and that you have the data base manager opened before you try the examples.

5.1 What you will achieve

When you have completed this exercise, you will have a complete application allowing you to navigate through the ORDERENT tables. The transitions that you will be able to do are:

1. From a list of all orders to an entry panel for a particular order. The order list will include quick view data for the customer (name and address information).

2. From the list or entry panel for an order, to the entry panel for the customer using quick view.

3. From the list or entry panel for an order, to the list panel for all customers using prompt.

4. From a list of all orders to a list of line items a particular order. The focal data will be for that particular order, and will include the quick view data for the customer.

5. From a line item, to the entry panel for the item catalog for the item.

6. From a line item, to a list of all the entries in the item catalog, using prompt.

7. From the list panel of customers, to all the orders for a particular customer.

5.2 Setting up the Persistence Builder Model.

You can begin the Persistence Builder model by importing the schema from the database (orderent) and then generating a model from the schema (use Orderent for the schema and model name). Before generating the model make sure you remove any underscores from the table names and attribute names in the schema. Once you have done this, the following model relationships need to be set up:

1. A ruler to subobject relationship from Orders to Lines. The role of Orders should be ruler, and the role of Lines is subobject. Note that the "ruler" association replaces the order number attribute in Lines.

2. A quick view relationship from Customers to Orders. The role of Customer should be qvsource, and the role of Orders is qvtarget. Note that the "qvsource" association replaces the customer number attribute in Orders.

3. A quick view relationship from Lines to Catalog. The role of Catalog should be qvsource, and the role of Lines is qvtarget. Note that the "qvsource" association replaces the item number attribute in Lines.

Make sure that primary keys are set up as follows:

1. Orders

order number

2. Lines.

ruler, line number

3. Customer

customer number

4. Item Catalog

5. item number

You must now update the schema and map to reflect these changes. In the schema you must add foreign key relationships for the associations. You need a foreign key relationship from customers to orders (linked by customer number), one from orders to lines (linked by order number) and one from item catalog to line items (linked by item number). The next step is to go the the map (this was generated along with the model from the schema) to reflect the changes you made in the model. You will see that the removed attributes show up as broken. You need to delete them. You will need to map the associations to the appropriate foreign key relationships. You can then generate the Persistence Builder model and service e classes. When you generate the model, make sure you use EADPEntityBeanImpl as the base class. This is the link between EADP and the classes generated by Persistence Builder.

A note for more experienced Persistence Builder users -- if you are familiar with Persistence Builder will you find it more natural to set up the foreign key relationships in the schema before generating the model. The advantage of this is that it avoids creating attribute names in the model which later need to be deleted. If you set up the foreign key relationships in the schema correctly, the model will be generated with all the ruler to subobject and quick view relationships. Of course, you will have to rename them as specified above, so that EADP can recognize them. This approach will also minimize the amount of rework you need to do to the map.

5.3 Setting up a test application.

1. In the VisualAge® Workbench, create a new project (call it OrderEntryProject), and a new package for the project (OrderEntryPackage).

5.4 Creating the data base parts

The next step is to create the data base parts to manage the tables and transitions. We'll start with a very small complex object structure; from a list of orders to a list of line items for that order. However, we will also add quick views to the customer and the item catalog.

The Java version of EADP is not quite as seamless as the Smalltalk version because it relies strictly on Java bean customization techniques (and cannot automatically create new classes). There are also differences in the way classes know about other classes in the system that must be allowed for. To compensate for this, the definition of the database classes is a two step process. The first step is to define the class names and how they participate in the complex object structure (what table the class maps to, and what its ruler is). The next step is to create the class and do any additional customizations to provide quick views and derived fields.

5.4.1 Setting up the database definition class

The overall complex definition for the application is defined by customizing a child of EADPDatabaseDefinition.

1. In the OrderEntryPackage, create a new class OrderentDefinitionClass with EADPDatabaseDefinition as its parent.

2. Open the Visual Composition for OrderentDefinitionClass and add two beans

    1. A bean called directory of type EADPDirectoryClass. Connect its "this" property to the currentDirectory property of the definition class.
    2.
    3. A bean called connection of type EADPVapConnection. Connect its "this" property to the vapConnection property of the definition class.

The overall application definition is now achieved by customizing the properties in these two beans.

3. The connection bean is customized with connection information. Customize the datastore property to point to the singleton for the datastore class generated out by Persistence Builder. For example, if you keep the default database name of Orderent, and use all defaults, you would enter

```
OrderentModelPackage.Services.OrderentOrderentDataStore.singleton
()
```

This step is the link that binds the EADP classes to the Persistence Builder classes. The Persistence Builder datastore class has information that lets it access all the other classes generated by Persistence Builder. EADP will use this in the next step to generate initialization strings that EADP uses in its operations.

4. Save the customizations (generate runtime code) before the next step. This is so that the custom editor for the definition bean can connect to the Persistence Builder classes.

5. The complex object definition is set up by customizing the complexObjectStructure property of the directory bean.

5.4.2 Customizing the complexObjectStructure property

When this part is customized, a custom edit panel appears that will define to EADP the overall complex object structure for the application. The function of this panel is to map application classes (children of EADPApplicationClass) to nodes in the complex object structure (each node has an associated table, and possibly a ruler table as well). Each node has information that EADP

needs at run time (and in some cases build time) to process the Persistence Builder relationships.

There is very little for you to do to accomplish this step; however, it triggers a great deal of system activity. What is happening is that the code generated by Persistence Builder is being analyzed to determine the relationships of interest to EADP. These are then bound into initialization strings for the complex object structure, so that the process does not have to be repeated at runtime. Note that it you go back into Persistence Builder to change your data model (which is likely to happen many times during the course of development) you have to repeat this step to make sure that EADP understands the Persistence Builder code.

5.4.2.1 Connecting to the database.

The first step is to use the Connect button. This will connect to the database, and initialize the other fields. Make sure that the Datastore that appears at the top left of the panel is correct. This step uses the datastore information you set up when the connection bean was customized, so make sure you did this and saved the customizations.

Once the connect button has processed, you will see a list of table names in the table list. Selecting a table will display the corresponding application class name EADP will use for that table. You have to create the classes with this name. You can come back here for reference if you forget the correct name (you may find it easier to browse the generated initialization strings, which will contain the class names as the first entry. These can be copied and pasted into the class name needed to define the application class when it is created within VisualAge).

At this point all customizations are complete. Press the OK button and save and close the definition class. Make sure you do this step; it is required to properly generate the initialization strings that bind the EADP and VAP classes.

5.4.3 Creating the Customers part

1. In the Workbench, define a new class CustomersApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type OrderentDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefintion property of the application class.

4. In the visual editor, add a bean called manager of type EADPDAManager.

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

5.4.4 Creating the Item Catalog part

1. In the Workbench, define a new class ItemcatalogApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type OrderentDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefintion property of the application class.

4. In the visual editor, add a bean called manager of type EADPDAManager.

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

Note -- if you are lazy you can copy the CustomersApplicationClass instead!

5.4.5 Creating the Orders part

1. In the Workbench, define a new class OrdersApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type OrderentDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefintion property of the application class.

4. In the visual editor, add a bean called manager of type EADPDAManager.

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

Note -- if you are lazy you can copy the CustomersApplicationClass instead!

Save this much of the customization before the next step, so that the quick views editor can connect to the database.

We have enough information set up at this point to make the quick view links to the Customer table. Quick views are now defined by customizing the quickViews property in the manager bean.

1. Customizing this property will place you in the quick views custom editor.

2. From the drop-down for Source Columns select qvsource.

3. You should see the columns for the customer table appear in the "QV Table Columns" list.

4. Select NAME from the "Table Columns" list. Press the "Add QV Column" button, and NAME will be added to the "Quick View Columns" list.

5. Repeat this for any other columns you want to pull into the quick view. In particular, you will want to add c_numb.

6. Press the Update button to save your selections and exit the "Create Quick View" panel by pressing the OK button.

When you return to the Visual Editor, save the part and exit.

You are probably wondering what the Zoom buttons do. The sample application is too simple to demonstrate them completely -- if Customers had quick view relationships itself, the "Zoom In" button would let you select one of those relationships, and then add its fields as quick views. "Zoom Out" lets you move back down. You can only update when you have Zoomed out to the original level.

5.4.6 Creating the Line Items part

1. In the Workbench, define a new class LinesFromOrdersApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type OrderentDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefintion property of the application class.

4. In the visual editor, add a bean called manager of type EADPDAManager.

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

We have enough information set up at this point to make the quick view links to the Catalog table. Quick views are now defined by customizing the quickViews property in the manager bean.

1. Customizing this property will place you in the quick views custom editor.

2. From the drop-down for Source Columns select qvsource.

3. You should see the columns for the catalog table appear in the "QV Table Columns" list.

4. Select DESCRIPTION from the "Table Columns" list. Press the "Add QV Column" button, and DESCRIPTION will be added to the "Quick View Columns" list.

5. Repeat this for any other columns you want to pull into the quick view.

6. Press the Update button to save your selections and exit the "Create Quick View" panel by pressing the OK button.

5.4.7 Finishing touches

5.4.7.1 Column Names

The default for the names of columns (as they appear in the table parts created by doing a quick form), is the data base name for the columns, separated into words and placed in mixed case (this is what VisualAge® uses also). However, EADP allows you to set up the column names globally. This is done by customizing the externalNames property for the directory bean in the definition class (for our example, OrderentDefinitionClass).

To define external names for the internal class and columns, customize this property. This will bring up the custom edit panel for external names.

1. A list of the internal classes you have defined so far will appear. OrdersApplicationClass,

A list of the columns you have defined (including the quick view and subobject columns) will

appear.

2. Type in an external name for the class (e.g "Orders") and press the "Set Application" button.

3. Type in an external names for the some of the columns and press the "Set Column" button after each one.

    1. C_NUMB "Customer Id"
    2.
    3. C_NUMB:NAME "Customer Name"

5.4.7.2 Object Names

There are times that EADP needs to display the name of an internal class (for error messages, and also to show which subobjects can be selected for a particular ruler object). The default is to use the internal class name (e.g. OrdersApplication). As explained above, you can set the external name for the object using the facilities of the external names custom editor. For example, you can specify "Line Items" as the external name for LinesFromOrdersApplicationClass.

5.4.7.3 Derived fields

Derived fields allow you to display information that is contained in a combination of columns, or spread over several tables. This data can later be used to drive business logic, just as if it existed directly on the database. We will add two cost fields; one to show the cost of each line item, and a second to show the total cost of an order.

To add the cost of a line:

1. Open LinesFromOrdersApplicationClass.

2. Open the properties sheet for the manager bean.

3. Select the computedColumns property to customize. This will bring up the computed columns custom editor.

4. There will be no entries in the Computed Columns list since you have not defined any computed fields yet.

5. From the Source Columns list select QUANTITY. Press the Set button next to "a Column".

You will see it appear in the "a Column" position.

6. From the Source Columns list select PRICE Press the Set button next to "b Column". You will see it appear in the "b Column" position.

7. You are now ready to write the formula to calculate the derived field. The rules for writing a formula are as follows:

    1. The source values are a, b, c, or d. You indicate which columns provide source values by adding them to the SOURCE COLUMNS list.
    2.
    3. Any valid Java expression involving +,-,/ and * can be used within the formula. You can use parentheses and literal numeric expressions (e.g. 12*(a+B)).

8. In our case, the derived field is just the product of the two source fields, so the formula is:

```
a*b
```

9. Enter the name linecost in the Computed Column Name text area.

10. Press the Update button.

11. Use the OK button to close the custom editor. Save and close the part.

You can also use string expressions in computations. String expressions are distinguished by the fact that they are enclosed in square brackets. Within string expressions, literals can be specified by enclosing them the "pointy brackets " (< and >). You need do do this if you are using a reserved symbol (a, b c, d, +, -, /, *, or a bracket) as a literal. The same operators (+, -, / and *) are used for string operations, but they have different meanings. A + is used to concatenate two strings. The other operators are used for substring operations. a-b means remove the string b from a. a*b is the string a up to the first occurrence of the substring b. a/b is the part of string a beyond the first occurrence of b. a%b is the part of a beyond the last occurrence of b.

The use of curly brackets around an expression forces its results to be evaluated as numeric. Here is an example -- suppose Campaigns are numbered CP-1, CP-2 etc. Tactics within a campaign are numbered T-1-1, T-1-2 etc. so for example the tactics for CP-35 are T-35-1 and T-35-2. We want to calculate the next tactic number. The first step is to add a quick view for campaign number to get the "a" column (which will have the value CP-35). The next step is to set up a summary column for the maximum existing tactic number (the "b" column, which will have the value (T-35-2). The formula for the next id number is now:

```
[T+[a/<->]+[<->+{[b%<->]+1}]]
```

Now we can add a summary field ordercost to orders.

1. Open OrdersApplicationClass.

2. Open the properties sheet for the manager bean.

3. Select the summaryColumns property to customize. This will bring up the summary columns custom editor.

4. Select LinesFromOrdersApplicationClass from the Subobject list (if you customized this to have an external name, the external name (e.g Line Items) will show up here). You should see the columns for the line item table appear in the "Source Columns" list.

5. Select linecost from the "Source Columns" list.

6. Select "Sum" from Summary Function List.

7. Specify the name for the summary column (ordercost) in the summary column name field.

8. Press the Update button. be added to the "Quick View Columns" list.

9. Press the Update button.

10. Use the OK button to close the custom editor. Save and close the part.

5.5 Setting up a default for the Line Item key.

EADP provides a mechanism to provide default values over "bound" relationships (such as a ruler or a "strong" quick view. Note that since the default is applied as a new object is created the relationship must already be in place before then. This is true if the new object is created within a subobject list (the ruler will already be established) or from a list of usages for the strong quick

view.

In this case we will default a new line number for an order by taking the maximum of the existing line numbers and adding one. This is a relatively simple calculation because only numbers are involved. However, as the discussion of computed values indicated, you can mix string and numeric values for a calculation of this type.

A default editor is defined by creating a child of EADPBusinessEditor. This class implements java.beans.PropertyEditor, so it allows you to use the BFT as a special editor. It is customized using a custom editor for the value manager (see the example below). Once you have set up the default editor, you link to a field by defining it as the special editor for that field.

5.5.1 Setting up a new summary column.

The default editor will use a new summary column called linemax, which is set up as above, but using the Maximum function to get the maximum line number.

5.5.2 Setting up a default editor.

1. In OrderentPackage, create a new class called LineNumberEditor that inherits from EADPBusinessEditor.

2. In the Visual Composition editor, add a bean of type OrderentDefintionClass (call it dataDef) and connect its "this" attribute to the dynaBeanDataDefinition attribute of the part.

3. Save the part, close and reopen it.

4. In the Visual Composition editor, add a bean of type EADPBusinessValueManager (call it manager) and connect its "this" attribute to the manager attribute of the part.

5. Save and close the part, then reopen it.

6. Open the properties sheet for the manager bean, select the businessValue property property, and double click to open the custom editor.

Enter the following formula:

```
&ruler;:summary:linemax&+1
```

The expression after the & sign will be translated to be the "a" column for the formula.

5.5.3 Making the default a special editor for an attribute.

1. In the Visual Composition editor for LinesFromOrdersApplicationClass open the properties sheet for the manager bean.

2. Open the custom editor for the specialConverters bean.

3. From the Table Columns list select lineordernumber.

4. In the Special Converter field type
```
OrderentPackage.LineNumberEditor
```

5. Press the Set button.

6. Close the special editor and the properties sheet and save the class.

5.6 Creating the applet to enter application.

EADP provides default parts to view the application. All you need to do to get started to set up an "application entry" applet. After that, you can customize individual panels as you desire. Most of the testing of the application logic can be done using just the application entry applet and the default panels.

5.6.1 Customizing the application applet.

1. In the Workbench, select the OrderEntryPackage, and create a new class OrdersEntryApplet as a child of EADPApplicationEntryApplet. The parent class will show up as a visual bean in the Visual Editor. The new class is defined by customizing properties of the parent bean.

2. Customize the databaseDefinition property to
```
new OrderentDefinitionClass()
```

This is the one visual customization that MUST be done to tie the visual parts to the underlying

application classes (everything else can be allowed to default).

5.6.2 Allowing the connection to be changed.

You specified a connection (probably the DB2 application driver and the local URL) when you set up the classes for the application within Persistence Builder. However, as you deploy the application you may want to change the driver, the URL, or the name of the database. The applet allows the URL and connection to be specified as parameters, and the userid and password can be entered on the applet screen.

However, in order to get this new connection information to be recognized, you need to make the Persistence Builder datastore adjustable. This requires a little more coding than most steps in EADP.

1. In the Workspace, select the services package that was generated. If you took all defaults it will be called OrderentOrderentPacakage.Services.

2. Select the data store class (it should be called OrderentOrderentDataStore), and open it.

3. Go to the BeanInfo tab and add a new property called connectionSpec of type DatabaseConnectionSpec. Note that this step will replace the connection information generated by VAP. If you want to retain the old information as a default, before you create the new property, copy the exsiting getConnectionSpec method to a new method called getBaseConnectionSpec. Then, after the new getConnectionSpec method is generated for the property, modify it to call getBaseConnectionSpec as the default initialization. Note that if you regenerate the VAP code, you will lose the getConnectionSpec method. You can replace it easily from the repository.

4. Now go to the Hierarchy tab and open the class definition. Add the following at the end of the first line

```
implements com.ibm.eadp.vap.EADPAdjustableDatastore
```

5.6.3 Setting the class path.

Before you can test the part, you must set up the class path. Select the Run option in the Workspace, then Check Class Path. You need EADPAccessClasses and VisualAge Persistence as well as whatever project you set up for the DB2 classes. If you placed the Persistence Builder classes generated for the application in a different project, make sure that is included as well.

5.6.4 Testing the part.

To test it, select the Run option and select Run as Applet or Run Main. Press the Connect Button. Once you are connected, you will see a list of the top objects. You can select one, and press the Open button. Testing of the various list and entry panels is described below; however, you can use the default panels if you want to.

5.7 Creating the list parts for the application.

Now that you have defined the data base parts, you can use visual programming techniques to create the user interfaces. EADP provides true container parts with tabular update capability for visual presentation of the lists.

You can skip this step if you want to, since EADP provides default parts which will give an immediate presentation of the application. However, for a real application you would probably want to do some customization.

We will begin by creating the visual part just for the ORDERS table, and then add the transition to the list of Line Items.

5.7.1 Creating the list part for Orders.

Orders is a top object. When a its list panel is invoked, the manager class and the connection are already set up. The only customization that needs to be done is to select which columns to display, and also which fields to show in the focal data.

The top object lists are brought up as new frames. To accomplish this, a visual bean EADPTopobjectForm (which inherits from panel) is provided. Each new top object list inherits from Frame, adds a bean of type EADPTopobjectForm, and implements the EADPListPanel interface.

1. In the Orderent package, create a new class OrdersListPanel as a child of Frame, implementing EADPListPanel.

2. In the Visual Editor, add a bean called TopobjectForm of type EADPTopobjectForm.

3. Implement the showListPanel method as follows:

```
getTopobjectForm().setDataManager(aManager);

this.show();
```

4. Now customize the listPanelClassName property of the manager bean in OrdersApplicationClass to

```
OrdersListPanel
```

This is how the data manager knows a special subobject list panel has been set up.

5. You can specify which columns you want to see on the list and the column lengths (in pixels) by customizing the Display Columns and Display Column Widths properties. For example, you can customize Display Columns as follows: (for ODBC)

```
onumb

qvsource:cnumb

status

qvsource:name

summary:ordercost
```

(for db2)

```
ordernumber

qvsource:customernumber

status

qvsource:name

summary:ordercost
```

Note that the summary column has a prefix, and that the quick view columns are prefixed by the name of the column they are linked to.

You should specify a length for every column you define in Display Columns by adding a corresponding entry to Display Column Widths.

6. Save the part. To test it, select the application entry applet (OderentEntryApplet) in the Workspace, and select the run option. Make sure that you have the class path set up properly

(you need EADPAccessClasses and VisualAge Persistence as well as whatever project you set up for the DB2 classes).

When the window comes up, press the Connect button, then the Open button with the Orders table selected. This will bring up the list panel for orders.

7. Now press the List Actions button, and select the Open option. You should see a list of all ORDERS.

You can use the selection text area to limit selection of the orders (this adds SQL selection criteria, so you should use real column names for real columns (e.g. order_number, customer_number, and status). This is function that EADP adds on top of Persistence Builder to allow limiting of the selection of top objects.

You can test the Open Row and Drill Down actions now to see what the default panels look like. In the next step we will set up a customized list panel for line items.

5.7.2 Creating the list part for Line Items.

Line Items is a subobject. When a subobject list panel is invoked, the manager class and the connection are already set up. The only customization that needs to be done is to select which columns to display, and also which fields to show in the focal data.

The subobject lists are brought up as new frames. To accomplish this, a visual bean that inherits from panel EADPSubobjectForm, is provided. Each new subobject list inherits from Frame, adds a bean of type EADPSubobjectForm, and implements the EADPListPanel interface.

1. In the Orderent package, create a new class LinesFromOrdersListPanel as a child of Frame, implementing EADPListPanel.

2. In the Visual Editor, add a bean called SubobjectForm of type EADPSubobjectForm.

3. Implement the showListPanel method as follows:

```
getSubobjectForm().setDataManager(aManager);

this.show();
```

4. You can customize the columns that are shown on the list by customizing Display Columns and Display Column Widths as for the orders list panel. One thing to watch out for is that the computed column (linecost) is known internally as computed:linecost.

5. There is also a section above the list that shows focal data. By default, all fields from the ruler (or rulers) are shown here. You can limit this by customizing the Display Fields property.

6. Save the part. You will have to wait until the next step to test it.

7. Now customize the listPanelClassName property of the manager bean in LinesFromOrdersApplicationClass to
`LinesFromOrdersListPanel`

This is how the data manager knows a special subobject list panel has been set up.

5.7.3 Testing the Orders and Line Items list parts.

Now we test the connection between the ORDERS and LINE_ITEMS tables. This will allow you to select one row from the ORDERS table, and open a list of Line Items for that order.

1. Go back the TestOrdersExternal and run the part.

2. Select one of the orders. The row will turn a different color (the row selection color) and the column a different color still (the cell selection color). These are customizable properties of the EADPTopObjectDisplay bean.

3. The dropdown list of subobjects should show "Line Items" (or LinesFromOrdersApplicationClass if you didn't set the external name) as the selected subobject.

4. Press the "Drill Down" button.

5. You should see a list of Line Items for that customer, and the order information in the focal data. You can select another order and open its Line Items also; another window will appear. Note that VisualAge® seems to like to put all the new windows at the same spot on the display, so you may have to move them off one by one to see the next one that comes up.

5.7.3.1 Testing customizations.

You can now test any customizations you have made (see 5.4.5, "Creating the Orders part"). Any changes you made to the column names for display should show up as the titles for the column

headings.

1. Open a list of Orders.

2. Tab to a cell under the C_NUMB column. It should turn a different color to show that is has been selected.

3. Move the mouse into the list table and press the right button. You will see a popup menu.

4. Click on the "Prompt" button.

5. You will see a list of customers (the list is not filled in yet). Enter a selection text (e.g NAME LIKE Joe%) and press the Open button. Now choose the row, and press the Select button. The corresponding row in the orders table will be updated with the selected customer.

5.7.4 Entry Panels.

All updates can be made using the list panels described above. However, for larger rows, entry panels may be required as well. EADP provides a convenient mechanism for setting up the entry panels and linking them to the list panels. We will demonstrate this by setting up entry panels for Orders and Line Items.

There are two types of entry panels that can be created. The first (a "default" entry panel) allows you to select which database columns (or added derived columns) you want to display as in the list panels, but it does not allow much flexibility on arranging the fields. The second technique lets you add and arrange label and text fields using the Visual Composition editor -- this allows you to make the panel look exactly the way you want it.

The technique for creating default entry panels is very similar to that for subobject panels

5.7.4.1 Creating the default entry part for Orders.

1. In the Orderent package, create a new class OrdersEntryPanel as a child of Frame, implementing EADPDisplayPanel.

2. In the Visual Editor, add a bean called EntryForm of type EADPEntryRow.

3. Implement the showDisplayForm method as follows:

```
getEntryForm().setTargetRow(eadpObject);

this.show();
```

4. You can customize the columns that are shown on the panel by customizing Column Names and Column Lengths as for the orders list panel.

5. Save the part. You will have to wait until the next step to test it.

6. Now customize the entryPanelClassName property of the manager bean in OrdersApplicationClass to
`OrdersEntryPanel`

This is how the data manager knows a special subobject entry panel has been set up.

5.7.4.2 Creating a custom entry part for Orders.

You may want to arrange fields more densely that the default entry panels allows (or you may want to have labels above instead beside the entry fields). A custom entry panel allows you more flexibility, while still retaining most of the benefits of using EADP. The custom panel, like the default panel, expects to be passed a target row. It uses three types of parts to handle data from the target row (a helper part, a specialized label part, and a specialized text field).

1. In the Orderent package, create a new class OrdersEntryPanelCustom as a child of Frame, implementing EADPEntryPanel.

2. In the Visual Editor, add a panel bean inside the frame.

3. In the Visual Editor, add a bean called DisplayHelper of type EADPDisplayHelper (note that you add this outside the frame part).

4. Connect the "this" attribute of the panel part to the "sourcePart" attribute of the DisplayHelper.

5. Open the property sheet for the DisplayHelper and set application name to the full name (including the package name) of the application class for Orders (i.e. Orderent.OrdersApplicationClass).

6. Save the part (this is so that the next customizations will pick up a list of column names from the Orders class).

7. Now you can add and arrange label and text fields (all of these should be added to the panel part).

    1. Add a bean of type EADPLabel. Set the name to OrderLabel. Open the property sheet for the label field and open the custom editor for the columnName propery. Select ordernumber from the drop down list and press Select, then Okay.
    2. The reason you are doing this this way is so that the label will show the common external name that you specify for order number. If you wanted to use a different name, you would just use an ordinary label field.
    3.
    4. Add a bean of type EADPEntryTextField. Set the name to OrderText. Open the property sheet for the text field and open the custom editor for the columnName propery. Select ordernumber from the drop down list and press Select, then Okay.
    5.
    6. Repeat the same process for any other fields you want to add. Note that you can rearrange the labels and fields any way you want to.

8. Implement the showDisplayForm method as follows:

```
getDisplayHelper().setTargetRow(eadpObject);

this.show();
```

9. Save the part. You will have to wait until the next step to test it.

10. Now customize the entryPanelClassName property of the manager bean in OrdersApplicationClass to

```
OrdersEntryPanelCustom
```

This is how the data manager knows a special subobject entry panel has been set up.

5.7.4.3 Creating the default entry part for Line Items.

1. In the Orderent package, create a new class LinesFromOrdersEntryPanel as a child of Frame, implementing EADPEntryPanel.

2. In the Visual Editor, add a bean called EntryForm of type EADPEntryRow.

3. You can customize the columns that are shown on the panel by customizing Column Names and Column Lengths as for the orders list panel. One thing to watch out for is that the computed column (linecost) is known internally as computed:linecost.

4. You can also add focal data to the panel by adding a focal data bean. Add a bean call FocalData of type EADPFocalDataForm. You can customize which fields to show as in the line items list panel.

5. Implement the showDisplayForm method as follows:

```
getEntryForm().setTargetRow(eadpObject);

getFocalData().setTargetRow(

eadpObject.getCurrentDatamanager().getFocalDataRow());

this.show();
```

6. Save the part. You will have to wait until the next step to test it.

7. Now customize the entryPanelClassName property of the manager bean in LinesFromOrdersApplicationClass to

```
LinesFromOrdersEntryPanel
```

This is how the data manager knows a special subobject entry panel has been set up.

5.7.4.4 Test the entry panels.

1. From the list panel, select a row and press the Open Row button.

2. Updates are flagged as focus is lost in a cell or data entry field. As you update a field in the entry panel, you should see the same field updated in the list panel and vice versa.

5.7.5 Complex Object Actions.

You can now exercise the complex object support for copy and delete.

5.7.5.1 Adding a Line Item for an Order.

This will show how a line item is set up using the correct order number when is it added to subobject list.

1. Open the list panel for orders, and open the list of orders.

2. Select an order, and select the then select Line Items and press the Drill Down button.

3. On the list of Line Items window, click the New Row choice in the File menu.

You will see a new row added to the list. Note that the order number is set to match the order you selected to find the list of Line Items. Fill in the rest of the data and press the Apply button.

5.7.5.2 Copying Orders and Line Items.

This will show how all the Line Items for one order are copied to another order.

1. Open the list panel for orders, and open the list of orders.

2. Select an order (one which has line items), and press the Copy Row option from the List Actions button.

3. Set an order number for the new order, and then click Apply.

4. Now open the list of Line Items for the new order. They will be identical to the list of Line Items for the old order.

5.7.5.3 Deleting Orders and Line Items.

This will show how all the line items for an order are deleted when that order is deleted.

1. Open the list panel for orders, and open the list of orders.

2. Select an order (one which has Line Items).

3. For the selected order, click the Delete Row option on the File Menu, then click Apply.

4. You can set up a standard list of line items using the data access builder to verify that the line items are really gone.

6.0 Setting up a version control sample

The next example uses the sample data base EADPSAMP which is shipped with the EADP product. Make sure you have this data base installed, and that you have the data base manager opened before you try the examples.

The samples can be loaded using either DB2 (UDB) or dBase V.

1. If you are using DB2 you must create the database definition in DB2. connection for the database.

2. If you are using ODBC, make sure that EADPSAMP is defined as a data source for dBase V.

3. If you are using ODBC, make sure that you set the shortVersions property in the datastore bean of the database definition class to True.

To install the samples, run the applet com.ibm.eadp.samples.EADPVersionSampleDisplay. Select the type of database manager you are using (DB2 or dBase 5) and then press the Create button.

6.1 What you will achieve.

This example will introduce some of the basic concepts of version control, and will demonstrate how EADP manages versions of data. It will familiarize you with the use of version selection objects, and how EADP chooses data at various versions.

The sample database for version control represents a small engineering records application that allows version controlled changes to part (item) and bill of material data. It has three tables:

1. AFFECTED_ITEM (VERSION for ODBC users)

This is the table for the version selection object. It is keyed by EC_NUMBER (EC_NUMB) and ITEM_NUMBER (I_NUMB). EC stands for "Engineering Change". In an engineering records application, an Engineering Change is used to bundle together a group of related changes to different pieces of data. It is an example of a persistent unit of work. The table includes a column VSO_SEQUENCE (VSOSEQ) which is used to select versions of item data.

2. ITEM_HEADER (ITEMHDR for ODBC)

This table contains basic information about the item. It is keyed by ITEM_NUMBER. The table includes two columns to enable version control, INSERTSEQ and EXTRACTSEQ (these are

called INSERT and EXTRACT for ODBC).

3. COMPONENT (COMPNNT for ODBC)

This table sets up the item to item relationship to establish a bill of material (the component items are parts of the assembly). The assembly item is the ITEM_NUMBER column, and the component number is the COMPONENT column (these are both key columns). The table includes two columns to enable version control, INSERTSEQ and EXTRACTSEQ.

6.1.1 Setting up the complex object structure.

1. ITEM_HEADER to COMPONENT

There is a simple complex object relationship from ITEM_HEADER to COMPONENT. The two tables are linked by ITEM_NUMBER, and COMPONENT_NUMBER is the additional key in the COMPONENT table.

2. COMPONENT to ITEM_HEADER.

A more subtle relationship is from COMPONENT_NUMBER back to the ITEM_HEADER data for the component item. We will implement this using quick views. One example will be a quick view to show the name of the component item along with the rest of the component data. As we will see, if the name changes from version to version, we will select the appropriate version of that data to display with the component. Later, we will use the quick view capability to add triggers for verifications of the component item's data when the component is processed.

3. AFFECTED_ITEM to ITEM_HEADER.

At first glance, it looks as if ITEM_HEADER should be the ruler of AFFECTED_ITEM, since the AFFECTED_ITEM table adds an additional key (EC_NUMBER). However, a more typical processing path is to first select a version of the item data (represented by the affected item), and then process the item data at that version. EADP allows you to do this by establishing the AFFECTED_ITEM table as a "VSO ruler" of the ITEM_HEADER table.

6.1.2 Setting up the Persistence Builder Definition.

Begin by importing the schema from the database (call it EADPSAMP) and generating the model from the schema. Note that the model includes root tables and version tables for both Item and Component.

There are three types of relationships that must be captured. The item is a ruler of the components. However, both the item and component have versions. In order to make the ruler ->

subobject relationship a true foreign key relationship, item root and component root tables are introduced. These have the keys of the tables "up to version" as their primary keys. The item root table has item number as its primary key, and the component table has bom number and component number as its primary key. The ruler -> subobject association is a foreign key relationship between the item number and the bom number. Next, there are version control related relationships. The item header has a version relationship with its root. The item header has the same keys as the root, plus the insert sequence number. The same sort of relationship is true for the component and its root. These are expressed as vaproot -> version relationships.

The affected item object acts as a version selection object for the items. It has two fields in its primary key: the item number, and an ec number (ec stands for engineering change, which is the way items are versioned in the real world). From a complex object perspective, the flow of control is from affected item to item header, so the affected item is defined in a ruler -> subobject relationship as the ruler of the item root. However, this is reversed from most ruler -> subobject relationships because the item key (the item number) forms part of the key of the affected item. Because the "single link" and "many link" classes have different Java types, this reversed relationship has a slightly different name for the roles (vsoruler and vsosubobject). This is so the generated code can be recognized properly by the EADP classes. Finally, there is a quick view relationship from the component number in the component back to the item root.

The net effect of all these relationships is that almost every attribute in the "data" objects gets replaced with an association. This is because the database is set up with very few data fields; in practice there would probably be twenty others that would be unaffected by all the model activity.

The following associations must be created:

1. A ruler to subobject relationship from the item root to the component root (the item root has the role of ruler and the component root has the role of subobject). This replaces the item number attribute in the component root (and forms part of the primary key).

2. A quick view relationship from the component root back to the item number (make sure the foreign key relationship maps to the component number in the component root). The item root has the role of qvsource, and the component is qvtarget. This replaces the component number in the component root and is the second part of the primary key.

3. A vsoruler to vsosubobject relationship from the affected item to the item root. The affected item has the role of vsoruler, and the item root is vsosubobject. The vsosubobject association replaces item number in the affected item and forms part of the primary key.

4. A version relationship from the item root to the item header. The item root has the role of vaproot and the item header has the role of version. The root association replaces item number in the item header.

5. A version relationship from the component root to the component. The component root has the role of vaproot and the component has the role of version. The vaproot association replaces item number and component number in the component.

Once you have these set up the classes should have the following primary keys:
1. Affected item.

vsosubobject, ec number.

2. Item root

item number

3. Item header

vaproot, insert sequence.

4. Component root

ruler, qvsource

5. Component

vaproot, insert sequence

Note that the combination root -> subobject -> version from item header to component is a fuzzy (many to many) relationship. You do not have to do anything more to establish this within Persistence Builder. EADP provides all the function necessary to cut through the versions.

You must now update the schema and map to reflect these changes. In the schema you must add foreign key relationships for the associations. You need a foreign key relationship from item root to items (linked by item number), one from item root to component root (linked by item number), one from item root to affected item (linked by item number), one from item root to component (linking item number in the item root to the component number in the component) and one from component root to component (linking the item number and component number). lines (linked by order number). The next step is to go the the map (this was generated along with the model from the schema) to reflect the changes you made in the model. You will see that the removed attributes show up as broken. You need to delete them. You will need to map the associations to the appropriate foreign key relationships. You can then generate the Persistence Builder model and service e classes. When you generate the model, make sure you use EADPEntityBeanImpl as

the base class. This is the link between EADP and the classes generated by Persistence Builder.

6.2 Setting up a version control sample application.

1. In the VisualAge® Workbench, create a new package (call it EADPSampPackage)

6.2.1 Setting up the database definition class

The overall complex definition for the application is defined by customizing a child of EADPDatabaseDefinition.

1. In the EADPSampPackage, create a new class EADPSampDefinitionClass with EADPDatabaseDefinition as its parent.

    1. A bean called directory of type EADPDirectoryClass. Connect its "this" property to the currentDirectory property of the definition class.
    2.
    3. A bean called connection of type EADPVapConnection. Connect its "this" property to the vapConnection property of the definition class.

The overall application definition is now achieved by customizing the properties in these two beans.

2. The connection bean is customized with connection information. Customize the datastore property to point to the singleton for the datastore class generated out by Persistence Builder. For example, if you keep the default database name of Orderent, and use all defaults, you would enter

```
EadpsampModelPackage.Services.EadpsampEadpsampDataStore.singleton
()
```

This binds the EADP classes to the Persistence Builder classes.

3. Save the customizations (generate runtime code) before the next step.

4. The complex object definition is set up by customizing the complexObjectStructure property of the directory bean.

6.2.2 Customizing the complexObjectStructure property

When this part is customized, a custom edit panel appears that will define to EADP the overall complex object structure for the application. The function of this panel is to map application

classes (children of EADPApplicationClass) to nodes in the complex object structure (each node has an associated table, and possibly a ruler table as well). At each node, the key columns are defined. The good news is that you don't need to supply any additional information here.

6.2.2.1 Connecting to the database.

The first step is to use the Connect button. This will connect to the database, and initialize the other fields. Make sure that the Datastore that appears at the top left of the panel is correct. This step uses the datastore information you set up when the connection bean was customized, so make sure you did this and saved the customizations.

Once the connect button has processed, you will see a list of table names in the table list. Selecting a table will display the corresponding application class name EADP will use for that table. You have to create the classes with this name. You can come back here for reference if you forget the correct name.

At this point all customizations are complete. Press the OK button and save and close the definition class. Make sure you do this step; it is required to properly generate the initialization strings that bind the EADP and VAP classes.

6.2.3 Creating the Affected Item part.

1. In the Workbench, define a new class AffectedItemApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type EadpsampDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefintion property of the application class.

4. In the visual editor, add a bean called manager of type EADPVsoManager (note that you are using EADPVsoManager instead of EADPDAManager here. EADPVsoManager is a child of EADPDAManager).

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

6. Set the vsoSequenceColumn property to vso_sequence if you are using DB2, and to vsoseq for dBase V.

6.2.4 Creating the Item Header part.

1. In the Workbench, define a new class ItemHeaderFromAffectedItemApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type EadpsampDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefinition property of the application class.

4. In the visual editor, add a bean called manager of type EADPDAManager.

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

6. Set the rulerIsVso property to True.

7. Set the versionControlled property to True.

6.2.5 Creating the Component part.

1. In the Workbench, define a new class ComponentFromItemHeaderFromAffectedItemApplicationClass as a child of EADPApplicationClass. Note that the name should match what was specified when you set up the complex object structure.

2. In the visual editor, add a bean called definition of type EadpsampDefinitionClass.

3. Connect the "this" property of the definition bean to the databaseDefinition property of the application class.

4. In the visual editor, add a bean called manager of type EADPDAManager.

5. Connect the "this" property of the manager bean to the currentDatamanager property of the application class.

6. Set the versionControlled property of the manager bean to True.

7. Save the part (you need to do this before the next step so that the part is linked properly with its definition class.

8. Now open the custom editor for quickView property:

    1. Select component_number as the source column.
    2.
    3. Select item_header as the quick view table.
    4.
    5. Select item_name as the quick table column.
    6.
    7. Press the Add Qv Column button.
    8.
    9. Press the Update Button.
    10.
    11. Press the OKAY Button.
9. Save the part again.

6.2.6 Adding Item data to the Affected Item Part.

The item number in the affected item was replaced by the vsoruler relationship. In order to see the item data, this has to be accessed through the vsoruler -> vsosubobject relationship. This can be done easily in EADP by defining the item data you wsnt to see as summary data using the summary function of FIRST.

1. In the Workbench, open the visual editor for the AffecteditemApplicationClass and open the property sheet for manager bean.

2. Open the custome editor for summaryColumns.

3. Select ITEMHEADERFROMAFFECTEDITEMAPPLICATIONCLASS from the list of subobjects.

4. Select vroot?itemnumber as the source column, and FIRST as the summary function.

5. Specify a name for the summary column (e.g. item#).

6. Press the Update button.

7. Repeat this to add the item name as a summary column.

8. Now press the OK button.

9. Save the AffecteditemApplicationClass bean.

6.3 Creating the visual parts.

The only visual part you need to create for the rest of the demonstration is an application entry applet. To do this, create a part called EADPSampEntryApplet that inherits from EADPAplicationEntryApplet. Customize it as follows:

1. Customize the databaseDefinition property to

```
new EADPSampDefinitionClass()
```

2. Save the bean.

3. Now update the class path to include the generated VAP classes, EADPAccessClasses and VisualAge Persistence as well as whatever project you set up for the DB2 classes.

6.4 Creating, Deleting and Undoing Versions

This section introduces some of the more advanced features of version control. Each new version allows you to isolate changes and protect the data you have entered at other versions. As long as the version is in informal status, you can back out the work you have done at that version by deleting the version or by using the undo function to reverse the changes to a particular object. These examples do not require you to build any new panels. Instead, they will guide you through functions that you would perform as a user of the application that you have already built.

If you want to restore the database, you can do this easily by reinstalling it.

6.4.1 Creating a New Version.

1. Run the ItemHeaderTopDisplay panel, connect, and open the list of affected items.

2. Select the row for Item P00001 at EC V00001.

3. Press "New Version" on the popup menu for the table part.

4. A new row will appear with all fields filled in except the EC number. Enter V00003 in this field (be sure to move the cursor off the field so the update is recorded).

5. Select "Apply" from the popup list menu.

You can now use the new version to modify header data, or to add, change or modify component data. There are some specific updates will we make in the next sections to demonstrate the undo and delete functions.

6.4.2 Using the Undo Function.

The undo function is used to back out the changes made by a particular version. It is invoked automatically when the version is deleted, but you can also use it on selected rows to restore the data for just that row.

6.4.2.1 Undo of Modified Data.

The simplest undo is of data that was modified at the version.

1. On the ItemHeaderTopDisplay panel change the Name of Item P00001 at V00003 to Item One C. Be sure to move the cursor off of the updated cell before you apply the change.

2. Now open the list of components for P00001 at V00003 and make the following updates:

    1. Delete the component with component number P00003.
    2.
    3. Change the quantity of the component with component number P00004 to 3.
    4.
    5. Add a component with component number P00005.

3. Now undo the changes to the item header:

    1. Select the row for Item P00001 at V00003.
    2.

3. On the popup menu for the table part, click "Undo Row".
4.
5. On the popup menu for the table part, click "Refresh". You will see that the name has been restored to its original value, and that the insert and extract sequence numbers show that the data is no longer affected by V00003.
6.
7. Refresh the component list panel. Notice that it is unchanged. Undo does not cascade if modified data is removed.

4. Now undo the added component:

1. On the component list panel, select the component with component number P00004.
2.
3. Select "Undo Row" from the popup menu.
4.
5. Refresh the panel. Notice that the row is missing. Since the version added the row, undoing it removes the row completely.

5. Now undo the deleted component. To do this, we have to be able to see it again.

1. From the popup menu, select "Open for Undo". Notice that the deleted component appears. This is because it was extracted (not physically deleted). If the component had been inserted by V00003 and then deleted, undo would not be able to restore it.
2.
3. Select the deleted component, and press "Undo Row" on the popup menu.
4.
5. Refresh the list. The deleted row has been restored.

6.4.2.2 Undo of Inserted Data.

When undo is applied to data that was created by the version, that data is physically deleted. The processing is the same as if the row was deleted (using Delete Row). This will cascade the delete to any subobjects.

We do not have data set up to completely demonstrate how this works (it is likely that your application may never be complicated enough to fully exploit this feature). However, we can demonstrate delete cascade by deleting the second version of an item header. Before starting this exercise, be sure you have restored the data to its original state.

1. Run the ItemHeaderTopDisplay panel, connect, and open the list of affected items.

2. Select the row for Item P00001 at EC V00002.

3. Open the item header.

4. Open the subobject list for components at EC V00002. Do this before deleting the header data so that you can see how the subobjects are affected.

5. Select Delete Row from the list popup menu on the header list panel, then select apply.

6. Refresh both the header list panel and the component list panel

7. Note that component list panel is now empty.

8. Now select "Refresh for Undo" on the on the popup menu for the table part.

9. The components will all appear again. However, they cannot be restored here, because there is no version of header data at V00002.

6.4.2.3 Undo of Extracted Data.

This continues the previous example. When the undo action is executed for extracted data, the action cascades to restore any subobjects that were also extracted.

1. On the item header list panel, select "Refresh for Undo" from the popup menu.

2. The deleted row (item P00001 at EC V00002) should reappear.

3. Select the row, and "Undo Row".

4. Now refresh both the item and component list panels. Notice that the deleted data has been restored on both panels.

6.4.3 Deleting a Version.

Deleting a version removes all the affects of that version, and also deletes the version selection object for that version.

1. Add a new version V00003 to item P00001 (see 6.4.1, "Creating a New Version."). Use the new version to modify data for both the header and components.

2. Open the component list panel at version V0002. Notice that the components that you

modified at V0003 reflect this with their extract sequence numbers.

3. Now select item P00001 at V00003 on the header list panel, and select "Delete Version" from the popup menu.

4. Select Apply.

5. Refresh the component list panel at V00002. The components that were extracted by V00003 are now unextracted.

6. Refresh the item header list panel. The version V00003 is gone.

The version is completely removed. Refresh for undo will not restore it on the list panels.

7.0 Batch File Support

Templates can be used to set up batch interface files or reports. The example shown here will create a small report.

7.1 Creating Report Templates

The example given here is a very simple one. In a real situation, you might begin with an example of your output as a model for the template file. You can use the file options in the Template Editor to bring in a file to get started (or you can file out and file in the templates if you want to use a more powerful editor during the template creation).

Here are some basic steps to prepare a set of templates for a report:

7.1.1 Opening the Template Editor for Reports

Templates for an application are set up using the EADPTemplateDefinition bean. The first step is the add a bean of this type in the Visual Composition editor for OrderentDefinitionClass (call the bean part templateDef). Connect this "this" attribute of the templateDef bean to the templateDefinition attribute of OrderentDefinitionClass.

1. In the properties sheet for the TemplateDefinition bean, select templateDictionary, and double click to open the custom editor.

This will open the Template Editor. All templates are stored as entries in Java dictionaries. The

Template Editor allows you to edit these dictionary entries as if they were ordinary text files, and to file them in and out of sequential files.

2. You are now ready to create the templates needed for the report.

7.1.2 The Structure of a Template

A template is a piece of text with imbedded keywords. The keywords are delimited by dollar signs. For example:

text1.. $keyword1$ text2.... $keyword2$... etc.

A template can span multiple lines of text. However, a special keyword $+$ at the end of a line indicates that a new line character should not be added when the template is processed.

When a template is processed, the text outside the keyword is passed along as in into the output stream. The text inside a keyword is evaluated according to the following rules:

1. The text up to the first comma is checked to see if it matches the name of a VisualAge® class. If it does, the "macro" class method of that class is invoked. One of the passed parameters is the rest of the text for the keyword (this is parsed by each macro).

The macro we will be using for reports is EADPReportMacro. Its macro method expects the rest of the keyword string to be in the format "(template name),(internal class name)".

1. The template name must be provided. It indicates which template will be used to process the next stage of the report.
2.
3. The EADPReportMacro expects the "current row" to be passed as a parameter. If the internal class name is not included, it will process everything in the template based on the current row. If the internal class name is included, the class should be a subobject for the table for the current row. The current row is used to select rows of the subobjects, and then each of these is used to process the template. We will use this technique to start with a row from the customer table, and use it to select all orders for the customer. Then each row from the orders table will be used to select line items for that order.

2. If the first part of the text is not that name of a class, it should match the name of a variable is the substitution list. For reports, this means that it should match an attribute name for the current row being reported (these are the data base column names and any additional columns added through the Quick View functions). A quick way of finding out which names to use is to open the Visual Composition Editor for the class, open the properties sheet for the manager bean you added previously, and open the custom editor for computed columns. The source columns there gives a list of the column names you can use.

Any text after the comma is assumed to be formatting information:

1. The first piece is the name of the class which will do the formatting (for example PadRightFormat). If this is omitted, no special formatting is done. The data appears just as it would on the list or entry panels.
2.
3. If there is more data (delimited by a second comma) this is passed to the formatting class. For example, the length of the field is passed to PadRightFormat.

7.1.3 Creating a Template to Start a Report

The report will be initiated from the list of orders. You will test the report from the Scrapbook. The first step is to write a template to start processing.

1. In the Template Editor, enter the following in the Template Name box:

StartOrdersReport

2. In the Template Editor, enter the following in the Text Area box:

```
Title: Orders Report

$EADPReportMacro,OrdersLine,OrdersApplicationClass$
```

This template will put out the line "Title: Orders Report" at the head of the report. The keyword then invokes EADPReportMacro, passing the current row, and telling it to use the OrdersLine template. The macro method of the EADPReportMacro class will prepare a variable list based on the column data in the current row before processing the template.

3. Click the Update button. You should see StartOrdersReport added to the list of templates.

7.1.4 Adding a Template for Order Data

The next step is to add a template to process the order data. This will have substitution keywords for the order columns we want on the report, and a macro keyword to bring in all the line items the order.

1. In the Template Editor, enter the following in the Template Name box:

OrdersLine

2. In the Template Editor, enter the following in the Text Area box:

```
Order #: $ordernumber$  Status: $status$

Line Items for Order

$EADPReportMacro,LinesLine,LineitemsFromOrdersApplicationClass$
```

(if you are using dBase V)

```
Order #: $onumb$  Status: $status$

Line Items for Order

$EADPReportMacro,LinesLine,LinesFromOrdersApplicationClass$
```

This template will put out the information for the indicated fields, using the values in the current row. A keyword then invokes EADPReportMacro, passing the current row, and telling it to use the LinesLine template. Since LinesFromOrdersApplicationClass is included, the current row will be used to select line items matching the order. The macro method of the EADPReportMacro class will loop through the list of rows prepare a variable list based on the column data on each row before processing the template.

3. Click the Update button. You should see OrdersLine added to the list of templates.

7.1.5 Adding a Template for Line Item Data

The final step is to add a template to process the line items. This will have substitution keywords for the line item columns we want on the report.

1. In the Template Editor, enter the following in the Template Name box:

LinesLine

2. In the Template Editor, enter the following in the Text Area box:

```
Line #: $orderlinenumber$ Item: $itemnumber$ $+$

Qty: $quantity$  Price: $price$
```

(if you are using dBase V)

```
Line #: $lnumb$ Item: $inumb$ $+$

Qty: $quantity$  Price: $price$
```